网络空间安全系列丛书

工业和信息产业科技与教育专著出版资金资助出版

# 密码算法应用实践

北京云班科技有限公司
文仲慧 何桂忠 周明波 牛传涛 张海黎 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

## 内 容 简 介

信息安全对密码提出了越来越高的要求。作为密码学重要组成部分的密码体制，近十几年来取得了长足的发展。本书根据密码的加密原理，分为 4 章，即分组密码、Hash 函数、序列密码、P1363 公钥标准简介，通过图表的方式对每种加密算法的原理进行了展示，并给出了部分加密算法的伪代码，具有很强的实用性。

本书适合从事信息安全的人士阅读，也可作为相关专业的教材或教学参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

**图书在版编目（CIP）数据**

密码算法应用实践 / 文仲慧等编著. —北京：电子工业出版社，2019.9
（网络空间安全系列丛书）
ISBN 978-7-121-35390-1

I. ①密… II. ①文… III. ①密码算法 IV. ①TN918.1

中国版本图书馆 CIP 数据核字（2018）第 252765 号

责任编辑：田宏峰
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：15.75　　字数：352 千字
版　　次：2019 年 9 月第 1 版
印　　次：2019 年 9 月第 1 次印刷
定　　价：68.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：tianhf@phei.com.cn。

# 丛书编委会

编委会主任：

樊邦奎　中国工程院院士

编委会副主任：

孙德刚　中国科学院信息工程研究所副所长、研究员

黄伟庆　中国科学院信息工程研究所研究员

编委会成员（按姓氏字母拼音排序）：

陈　驰　中国科学院信息工程研究所正高级工程师

陈　宇　中国科学院信息工程研究所副研究员

何桂忠　北京云班科技有限公司副总裁

李云凡　国防科技大学副教授

刘　超　中国科学院信息工程研究所高级工程师

刘银龙　中国科学院信息工程研究所副研究员

马　伟　中国科学院信息工程研究所科技处副处长、副研究员

苏桂甲　海军研究院高级工程师

王　妍　中国科学院信息工程研究所高级工程师

王小娟　北京邮电大学副教授

王胜开　亚太信息安全领袖成就奖获得者、教授

文仲慧　国家信息安全工程技术研究中心首席专家

吴秀诚　中国互联网协会理事、盈世 Coremail 副总裁、教授

姚健康　国际普遍接受指导组专家委员、教授

张　磊　中国民生银行总行网络安全技术总管、高级工程师

朱大立　中国科学院信息工程研究所正高级工程师

# 前　言

目前，越来越多的重要信息是以电子的形式存储和交换的，信息安全对密码提出了越来越高的要求。作为密码学重要组成部分的密码体制，近十几年来取得了长足的发展，特别是进入 20 世纪 90 年代以来，单钥密码新体制大量涌现，公钥密码标准化得到了迅速的发展。由于目前难以找到一本系统、全面介绍各种密码算法的图书，为使有关研究人员能够及时掌握各种密码算法，了解密码算法的发展新动向，开阔视野、活跃思路，我们编著了《密码算法应用实践》。

本书共分 4 章。

**第 1 章为分组密码。**

分组密码是现代密码学中的一个重要研究分支，其诞生和发展有着广泛的实用背景和重要的理论价值。目前这一领域还有许多理论和实际问题有待继续研究和完善。这些问题包括如何设计可证明安全的密码算法、如何加强现有算法及其工作模式的安全性、如何测试密码算法的安全性、如何设计安全的密码组件等。

对于分组密码，早期的研究基本上是围绕 DES 进行的，推出了一些类似的加密算法，如 LOKI、FEAL、GOST 等加密算法。进入 20 世纪 90 年代，人们对 DES 的研究更加深入，特别是差分密码分析（Differential Cryptanalysis）和线性密码分析（Linear Cryptanalysis）的提出，使人们不得不研究新的密码结构。IDEA 加密算法打破了 DES 的垄断局面，随后出现了 SQUARE、SHARK、SAFER-64 等加密算法，从理论上给出了最大差分特征概率和最佳线性逼近优势的界，证明了密码对差分密码分析和线性密码分析的安全性。

AES 的征集掀起了分组密码研究的新高潮，15 个 AES 候选加密算法反映了当前分组密码设计的水平，也可以说是近几年研究成果的一个汇总。

本章介绍的分组算法可以分为 3 类：第一类是采用菲斯特（Feistel）结构的分组密码，包括采用非平衡 Feistel 结构的分组密码和可看成 Feistel 结构扩展的 SKIPJACK 加密算法，以及采用 Hash 函数和其他密码函数构建的基于 Feistel 结构的分组密码；第二类是采用非 Feistel 结构的分组密码；第三类是 AES 征集的 15 个候选的加密算法。

由本章给出的分组密码的加密算法可以看出，分组密码的基本编码思想是由一些简单的基本变换构成层变换，通过多层迭代来达到混乱和扩散的目的的，从而构成强的密码变换。虽然加密的理论和技术又有了许多新发展，提出了一系列新思想、新技巧，但各种算法的研究和改进都是万变不离其宗，都是根据使用需求和实现环境，以期使单位时间的计算取得更

好的安全效益，是权衡加密算法密度和计算时间开销的折中。

分组密码加密的新思想和新发展表现为：一是在体系结构上更灵活多样；二是在基本运算选择上更多变快速；三是非线性变换的理论基础更趋完善；四是层密钥生成方法变化纷呈；五是分组密码与其他密码函数的内在联系越来越清晰；六是在密码中设置陷门的研究已深入到分组密码。

**第 2 章为 Hash 函数。**

20 世纪 70 年代以来，随着网络的逐步普及，人类社会步入信息化时代。自此，电子文件大量出现，其安全性一开始就令人担忧。电子文件的安全问题之一是电子文件的完整性。所谓完整性，主要是指电子文件是否有部分改动、删除或插入。在社会客观需求推动下，经过包括密码学家在内的大批学者的共同努力，创造性地设计出了可满足消息完整性验证的密码算法——Hash 函数。

本章主要介绍了典型的基于 Hash 函数的消息压缩值算法。

**第 3 章为序列密码。**

序列密码也称为流密码（Stream Cipher），是对称密码算法的一种。序列密码具有实现简单、便于硬件实施、加解密处理速度快、没有或只有有限的错误传播等特点，因此在实际应用中，特别是在专用或机密机构中保持着优势，典型的应用领域包括无线通信、外交通信。

分组密码以一定大小作为每次处理的基本单元，而序列密码则是以一个元素（一个字母或一个比特）作为基本的处理单元。序列密码是一个随时间变化的加密变换，具有转换速度快、低错误传播的优点，硬件实现电路更简单；缺点是低扩散（意味着混乱不够）、插入及修改的不敏感性。分组密码使用的是一个不随时间变化的固定变换，具有扩散性好、插入敏感等优点；缺点是加解密处理速度慢、存在错误传播。

序列密码涉及大量的理论知识，提出了众多的设计原理，也得到了广泛的分析，但许多研究成果并没有完全公开。这也许是因为序列密码目前主要应用于军事和外交等机密部门的缘故。目前，公开的序列密码加密算法主要有 RC4、SEAL 等。

本章主要介绍序列密码体制以及由此反映出来的加密新思想，主要包括：基于 Fibonacci 序列和 CA 构建的乱源发生器，FISH 加密算法和 PIKE 加密算法以模 232 的 Fibonacci 序列为乱源，是乘同余序列的推广，CA 是移位寄存器（或级间模二加移位寄存器）的推广，可看成是一种特殊的自动机；采用择多走/停的新步进方式的密码体制，PIKE 加密算法和 A5 加密算法的控制信息可来自乱源序列的某些位，也可来自信息的奇、偶校验位；各种新颖的加密技术，如面向软件实现的 RC4 加密算法，与常见的基于线性反馈移存器的序列密码不同，以一个相对比较大的代替表为基础，在自身的控制下，通过表中元素的对换进行缓慢的变化，同时生成随机数序列；序列密码许多比较成熟的密码分析方法难以直接用于 RC4 的密码分析；与 RC4 加密算法类似的 ISAAC 加密算法，但它不用对换，而是采用移位和算术加；

TwoPrime 加密算法吸取了分组密码的加密方法，以线性和非线性变换交替复合，以块为单位进行加乱，每一周期生成 64 比特的乱数序列；WAKE 加密算法和 SEAL 加密算法是速度极快的序列密码，用 5 条指令即可生成 1 字节的乱数。

**第 4 章为 P1363 公钥标准简介。**

公开密钥加密（Public-Key Cryptography）也称为非对称加密（Asymmetric Cryptography），是密码学发展史上的一次革命。公钥密码体制的编码系统是基于数学中的单向陷门函数。更重要的是，公钥密码体制采用了两个不同的密钥，对在公开的网络上进行保密通信、密钥分配、数字签名和认证有着深远的影响。

本章简要地介绍了 P1363 公钥标准，主要包括 RSA 方案、共同密钥生成方案、数字签名方案和校验方法，以及密钥交换算法 KEA。

本书在编写过程中，参考了大量关于密码算法的论文和标准，在此表示衷心的感谢。

鉴于笔者水平有限，加之时间仓促，本书难免会有不足和疏漏之处，敬请广大读者批评指正。

作　者

2019 年 7 月

# 目　录


第 1 章　分组密码 …… 1
1.1　数据加密标准和分组密码的四种工作方式 …… 1
1.1.1　DES 加密算法的研制经过 …… 1
1.1.2　DES 加密算法 …… 1
1.1.3　DES 的工作方式 …… 7
1.1.4　DES 加密思想和特点 …… 9
1.2　Lucifer 加密算法（DES 前身） …… 10
1.3　NewDES 加密算法 …… 12
1.3.1　NewDES 加/脱密编制 …… 13
1.3.2　非线性函数 $f$ 的选取 …… 15
1.4　DES 加密算法的变型 …… 16
1.4.1　多重 DES 加密算法 …… 16
1.4.2　子密钥独立的 DES 加密算法 …… 16
1.4.3　S 盒可选的 DES 加密算法 …… 16
1.4.4　S 盒可变的 DES 加密算法 …… 16
1.4.5　CRYPT(3)加密算法 …… 17
1.4.6　广义的 DES（GDES）加密算法 …… 17
1.4.7　DESX 加密算法 …… 18
1.4.8　Ladder-DES 加密算法 …… 18
1.5　FEAL 加密算法 …… 19
1.5.1　层函数 $F$ …… 20
1.5.2　层密钥生成规律 …… 20
1.6　LOKI 加密算法 …… 22
1.6.1　LOKI89 加密算法 …… 22
1.6.2　LOKI91 加密算法 …… 25
1.7　GOST 加密算法 …… 28
1.8　BLOWFISH 加密算法 …… 29

1.9 Khufu 和 Khafre 快速软件加密算法……31
1.10 CAST 加密算法……33
1.10.1 CAST-128 加密算法……33
1.10.2 CAST-256 加密算法……36
1.11 ICE 加密算法……45
1.12 MISTY 算法……47
1.13 TEA 加密算法……51
1.14 RC5 和 RC6 加密算法……52
1.14.1 RC5 加密算法……53
1.14.2 RC6 加密算法……54
1.15 RC2 加密算法……55
1.16 MacGuffin 加密算法……57
1.17 SKIPJACK 加密算法……60
1.18 BEAR 和 LION 加密算法……65
1.19 基于伪随机函数和 Hash 函数构造分组密码……66
1.19.1 DES 加密算法……66
1.19.2 Phil Karn 加密算法……67
1.19.3 Luby-Rackoff 加密算法……68
1.19.4 Stefan Lucks 加密算法……68
1.20 IDEA 加密算法……70
1.21 REDOC-Ⅱ和 REDOC-Ⅲ加密算法……73
1.21.1 REDOC-Ⅱ加密算法……73
1.21.2 REDOC-Ⅲ加密算法……75
1.22 SAFERK-64 和 SAFER+加密算法……76
1.22.1 SAFERK-64 加密算法……76
1.22.2 SAFER+加密算法……79
1.23 3-Way 加密算法……81
1.24 SHARK 加密算法……82
1.25 SQUARE 加密算法……84
1.26 MMB 加密算法……87
1.27 XMX 加密算法……88
1.28 Madryga 加密算法……89
1.29 FBCP 加密算法……91

1.30 VINO 加密算法 ····· 93
1.31 MULTI2 加密算法 ····· 97
1.32 非线性奇偶电路密码体制 ····· 99
1.33 CRYPTON 加密算法 ····· 105
1.34 DEAL 加密算法 ····· 111
1.35 DFC 加密算法 ····· 114
1.36 E2 加密算法 ····· 118
1.37 FROG 加密算法 ····· 125
1.37.1 FROG 算法加/脱密过程 ····· 126
1.37.2 密钥设置 ····· 129
1.38 HPC 加密算法 ····· 131
1.38.1 符号、约定及常数 ····· 131
1.38.2 密钥设置 ····· 132
1.38.3 子密钥 ····· 134
1.39 MAGENTA 加密算法 ····· 145
1.40 MARS 加密算法 ····· 146
1.41 Rijndael 加密算法 ····· 154
1.42 Serpent 加密算法 ····· 157
1.43 Twofish 加密算法 ····· 161

**第 2 章 Hash 函数** ····· 169
2.1 安全压缩算法 SHA ····· 169
2.2 消息压缩值算法 RIPEMD-160 ····· 170
2.3 消息压缩值算法 MD2 ····· 173
2.4 消息压缩值算法 MD4 ····· 175
2.5 消息压缩值算法 MD5 ····· 177

**第 3 章 序列密码** ····· 181
3.1 A5 加密算法 ····· 181
3.2 RC4 加密算法 ····· 183
3.3 ISAAC 加密算法 ····· 184
3.4 FISH 加密算法 ····· 186
3.5 PIKE 加密算法 ····· 186
3.6 TWOPRIME 加密算法 ····· 187

3.7 密钥自动变化的字加密算法 WAKE……190
3.8 SEAL 加密算法……193
3.9 现代圆盘密码……195
3.10 HKM/HFX 传真加密系统……197
3.11 PKZIP 加密算法……202
3.12 CA 自动机在序列密码乱源中的应用……203
3.12.1 基本概念和定义……203
3.12.2 CA 乱数发生器……205
第 4 章 P1363 公钥标准简介……211
4.1 RSA 方案（IF/ES）……212
4.2 共同密钥生成方案……214
4.3 数字签名方案和校验方法……218
4.3.1 建立在 IF 之上的可逆方案……218
4.3.2 建立在 IF 之上的不可逆方案……221
4.3.3 建立在 DL 或 EC 之上的不可逆方案……223
4.4 密钥交换算法 KEA……226
附录 A *S* 盒……231
附录 B *E* 函数与加/脱密的伪代码……235
参考文献……239

# 第 1 章

# 分 组 密 码

## 1.1 数据加密标准和分组密码的四种工作方式

### 1.1.1 DES 加密算法的研制经过

DES（Data Encryption Standard）加密算法的研究历经 6 年的时间（1968 年—1974 年）。20 世纪 60 年代末，IBM 公司的一个研究组在菲斯特（Feistel）的带领下研制出了 Lucifer 密码。1971 年，IBM 公司请塔其曼（Tuchman）和迈耶（Meyer）在 Lucifer 密码的基础上进一步做研究，以增加密码强度。他们经过 3 年多的研究，试图破解 Lucifer 密码，通过搜索强有力的代替置换网络和密钥编制函数，完成了 DES 加密算法的设计和编制工作。

1976 年年底，DES 加密算法被采纳为联邦标准，自 1977 年 7 月 15 日起正式生效。1986 年，NSA 宣布停止执行 DES 计划。从 1988 年 1 月 1 日起，除了电子资金过户（Electronic Fund Transfer），不再批准政府部门使用采用 DES 加密算法的产品，但已批准的产品可继续销售和使用。美国安全局（National Security Agency，NSA）感到 DES 加密算法的广泛使用已使之成为众矢之的。

### 1.1.2 DES 加密算法

#### 1．DES 加密算法的框架

DES 加密算法如图 1.1.1 所示。输入明文块 $M_0$（64 比特），$M_0=m_0m_1\cdots m_{63}$，经初始置换 IP（见表 1.1.1）换位成 $M_0=m_{57}m_{49}\cdots m_6$。

将 $M_0$ 分成左右两个半块（32 比特），$M_0=(L_0,R_0)$，$(L_0,R_0)$经过第 1 层变换后记为$(L_1,R_1)$，其中，$L_1=R_0$，$R_1=L_0\oplus f(R_0,K_1)$。

经过第 $i$（$i$=1,2,…,16）层变换后记为$(L_i,R_i)$。其中，$L_i=R_{i-1}$，$R_i=L_{i-1}\oplus f(R_{i-1},K_i)$，此处的 $f$ 函数将在下面详述，$K_i$ 为密钥，经过 16 层变换后，预输出为$(R_{16},L_{16})$，再经 $IP^{-1}$ 置换成密文块 $C$。

#### 2．$f$ 函数

$f$ 函数是 DES 加密算法的核心部分（见图 1.1.2），进入第 $i$ 层时，$f$ 函数的输入 $R_{i-1}$（32

比特）先经扩张函数 $E$ 变成 48 比特，设 $R_{i-1}=r_0r_1r_2\cdots r_{31}$ 经 $E$ 扩张后记为 $T_{i-1}=E(R_{i-1})$。

$$T_{i-1}=t_0t_1t_2\cdots t_{47}$$

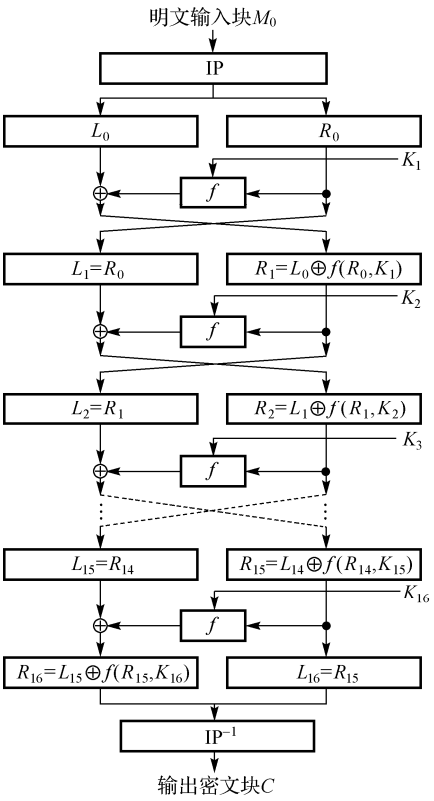



图 1.1.1　DES 加密算法

表 1.1.1　置换表 IP 和 $IP^{-1}$

| IP | | | | | | | |
|---|---|---|---|---|---|---|---|
| 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 |
| 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 61 | 53 | 45 | 37 | 29 | 21 | 13 | 5 |
| 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 |
| 56 | 48 | 40 | 32 | 24 | 16 | 8 | 0 |
| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 |
| 60 | 52 | 44 | 36 | 28 | 20 | 12 | 4 |
| 62 | 54 | 46 | 38 | 30 | 22 | 14 | 6 |

| $IP^{-1}$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| 39 | 7 | 47 | 15 | 55 | 23 | 63 | 31 |
| 38 | 6 | 46 | 14 | 54 | 22 | 62 | 30 |
| 37 | 5 | 45 | 13 | 53 | 21 | 61 | 29 |
| 36 | 4 | 44 | 12 | 52 | 20 | 60 | 28 |
| 35 | 3 | 43 | 11 | 51 | 19 | 59 | 27 |
| 34 | 2 | 42 | 10 | 50 | 18 | 58 | 26 |
| 33 | 1 | 41 | 9 | 49 | 17 | 57 | 25 |
| 32 | 0 | 40 | 8 | 48 | 16 | 56 | 24 |

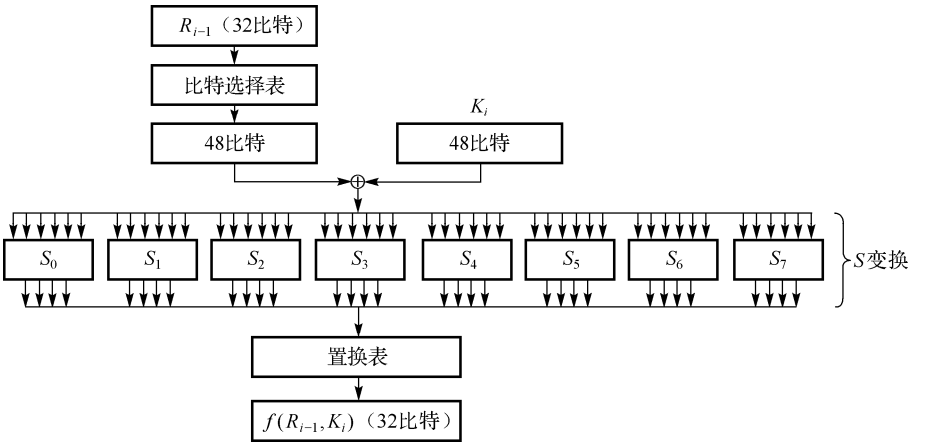



图 1.1.2　$f$ 函数框图

参考表 1.1.2 的比特选择表和置换表，其中 $t_0=r_{31},t_1=r_0,t_2=r_1,\cdots,t_{46}=r_{31},t_{47}=r_0$。$T_{i-1}$ 与 $K_i$ 的 48 比特按位进行模 2 加可得 $B=b_0b_1\cdots b_{47}$。$B$ 顺序地按 6 比特分组，分别进入代替表（$S$ 盒）$S_0,S_1,\cdots,S_7$，见表 1.1.3。

**表 1.1.2　比特选择表和置换表**

| 比特选择表 | | | | | |
|---|---|---|---|---|---|
| 31 | 0 | 1 | 2 | 3 | 4 |
| 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 11 | 12 | 13 | 14 | 15 | 16 |
| 15 | 16 | 17 | 18 | 19 | 20 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 23 | 24 | 25 | 26 | 27 | 28 |
| 27 | 28 | 29 | 30 | 31 | 0 |

| 置换表 | | | |
|---|---|---|---|
| 15 | 6 | 19 | 20 |
| 28 | 11 | 27 | 16 |
| 0 | 14 | 22 | 25 |
| 4 | 17 | 30 | 9 |
| 1 | 7 | 23 | 13 |
| 31 | 26 | 2 | 8 |
| 18 | 12 | 29 | 5 |
| 21 | 10 | 3 | 24 |

**表 1.1.3　$S$ 盒**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 14 | 04 | 13 | 01 | 02 | 15 | 11 | 08 | 03 | 10 | 06 | 12 | 05 | 09 | 00 | 07 | $S_0$ |
| 1 | 00 | 15 | 07 | 04 | 14 | 02 | 13 | 01 | 10 | 06 | 12 | 11 | 09 | 05 | 03 | 08 | |
| 2 | 04 | 01 | 14 | 08 | 13 | 06 | 02 | 11 | 15 | 12 | 09 | 07 | 03 | 10 | 05 | 00 | |
| 3 | 15 | 12 | 08 | 02 | 04 | 09 | 01 | 07 | 05 | 11 | 03 | 14 | 10 | 00 | 06 | 13 | |
| 0 | 15 | 01 | 08 | 14 | 06 | 11 | 03 | 04 | 09 | 07 | 02 | 13 | 12 | 00 | 05 | 10 | $S_1$ |
| 1 | 03 | 13 | 04 | 07 | 15 | 02 | 08 | 14 | 12 | 00 | 01 | 10 | 06 | 09 | 11 | 05 | |
| 2 | 00 | 14 | 07 | 11 | 10 | 04 | 13 | 01 | 05 | 08 | 12 | 06 | 09 | 03 | 12 | 15 | |
| 3 | 13 | 08 | 10 | 01 | 03 | 15 | 04 | 02 | 11 | 06 | 07 | 12 | 00 | 05 | 14 | 09 | |

续表

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 00 | 09 | 14 | 06 | 03 | 15 | 05 | 01 | 13 | 12 | 07 | 11 | 04 | 02 | 08 | |
| 1 | 13 | 07 | 00 | 09 | 03 | 04 | 06 | 10 | 02 | 08 | 05 | 14 | 12 | 11 | 15 | 01 | $S_2$ |
| 2 | 13 | 06 | 04 | 09 | 08 | 15 | 03 | 00 | 11 | 01 | 02 | 12 | 05 | 10 | 14 | 07 | |
| 3 | 01 | 10 | 13 | 00 | 06 | 09 | 08 | 07 | 04 | 15 | 14 | 03 | 11 | 05 | 02 | 12 | |
| 0 | 07 | 13 | 14 | 03 | 00 | 06 | 09 | 10 | 01 | 02 | 08 | 05 | 11 | 12 | 04 | 15 | |
| 1 | 13 | 08 | 11 | 05 | 06 | 15 | 00 | 03 | 04 | 07 | 02 | 12 | 01 | 10 | 14 | 09 | $S_3$ |
| 2 | 10 | 06 | 09 | 00 | 12 | 11 | 07 | 12 | 15 | 01 | 03 | 14 | 05 | 02 | 08 | 04 | |
| 3 | 03 | 15 | 00 | 06 | 10 | 01 | 13 | 08 | 09 | 04 | 05 | 11 | 12 | 07 | 02 | 14 | |
| 0 | 02 | 12 | 04 | 01 | 07 | 10 | 11 | 06 | 08 | 05 | 03 | 15 | 13 | 00 | 14 | 09 | |
| 1 | 14 | 11 | 02 | 12 | 04 | 07 | 13 | 01 | 05 | 00 | 15 | 10 | 03 | 09 | 08 | 06 | $S_4$ |
| 2 | 04 | 02 | 01 | 11 | 10 | 13 | 07 | 08 | 15 | 09 | 12 | 05 | 06 | 03 | 00 | 14 | |
| 3 | 11 | 08 | 12 | 07 | 01 | 14 | 02 | 13 | 06 | 15 | 00 | 09 | 10 | 04 | 05 | 03 | |
| 0 | 12 | 01 | 10 | 15 | 09 | 02 | 06 | 08 | 00 | 13 | 03 | 04 | 14 | 07 | 05 | 11 | |
| 1 | 10 | 15 | 04 | 02 | 07 | 12 | 09 | 05 | 06 | 01 | 13 | 14 | 00 | 11 | 03 | 08 | $S_5$ |
| 2 | 09 | 14 | 15 | 05 | 02 | 08 | 12 | 03 | 07 | 00 | 04 | 10 | 01 | 13 | 11 | 06 | |
| 3 | 04 | 03 | 02 | 12 | 09 | 05 | 15 | 10 | 11 | 14 | 01 | 07 | 06 | 00 | 08 | 13 | |
| 0 | 04 | 11 | 02 | 14 | 15 | 00 | 08 | 13 | 03 | 12 | 09 | 07 | 05 | 10 | 06 | 01 | |
| 1 | 13 | 00 | 11 | 07 | 04 | 09 | 01 | 10 | 14 | 03 | 05 | 12 | 02 | 15 | 08 | 06 | $S_6$ |
| 2 | 01 | 04 | 11 | 13 | 12 | 03 | 07 | 14 | 10 | 15 | 06 | 08 | 00 | 05 | 09 | 02 | |
| 3 | 06 | 11 | 13 | 08 | 01 | 04 | 10 | 07 | 09 | 05 | 00 | 15 | 14 | 02 | 03 | 12 | |
| 0 | 13 | 02 | 08 | 04 | 06 | 15 | 11 | 01 | 10 | 09 | 03 | 14 | 05 | 00 | 12 | 07 | |
| 1 | 01 | 15 | 13 | 08 | 10 | 03 | 07 | 04 | 12 | 05 | 06 | 11 | 00 | 14 | 09 | 02 | $S_7$ |
| 2 | 07 | 11 | 04 | 01 | 09 | 12 | 14 | 02 | 00 | 06 | 10 | 13 | 15 | 03 | 05 | 08 | |
| 3 | 02 | 01 | 14 | 07 | 04 | 10 | 08 | 13 | 15 | 12 | 09 | 00 | 03 | 05 | 06 | 11 | |

代替的规则如下：输入 $S_i$ 的 6 比特的左端 1 比特和右端 1 比特组成二进制数 0～3 中的某个数，作为取 $S_i$ 的行数；中间 4 比特组成二进制数 0～15 中的某个数，作为取 $S_i$ 的列数；在 $S_i$ 表行列交叉处取得一个数，用 4 比特表示作为 $S_i$ 盒的输出。例如，若输入 $S_0$ 盒的 6 比特为“011011”，则行数为“01”，列数为“1101”，即 13。$S_0$ 的第 1 行第 13 列的元素为 5，于是输出为“0101”，如下所示。

$S_0$ 盒

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 14 | 4 | 13 | 1 | 2 | 15 | 11 | 8 | 3 | 10 | 6 | 12 | 5 | 9 | 0 | 7 |
| 1 | 0 | 15 | 7 | 4 | 14 | 2 | 13 | 1 | 10 | 6 | 12 | 11 | 9 | 5 | 3 | 8 |
| 2 | 4 | 1 | 14 | 8 | 13 | 6 | 2 | 11 | 15 | 12 | 9 | 7 | 3 | 10 | 5 | 0 |
| 3 | 15 | 12 | 8 | 2 | 4 | 9 | 1 | 7 | 5 | 11 | 3 | 14 | 10 | 0 | 6 | 13 |

经过 $S$ 盒后的 32 比特再经过置换表可改变比特位置，结果作为 $f$ 函数的输出。

### 3．子密钥发生器

64 比特的初始密钥 $K$ 通过子密钥发生器变成 $K_1,K_2,\cdots,K_{16}$，分别作为 1 到 16 层的 $f$ 函数子密钥（48 比特）。

在 64 比特的初始密钥 $K$ 中，除去 8 比特（第 7、15、23、31、39、47、55、63 比特作为校验位），其余 56 比特送入置换 PC Ⅰ，经过坐标置换后分成两组，每组 28 比特，分别送入 $C$ 寄存器和 $D$ 寄存器中。在各次迭代中，$C$ 和 $D$ 寄存器分别将存储的数按移位次数表进行循环左移。每次移位后将 $C$ 和 $D$ 寄存器的第 6、9、14、25 比特除去，其余比特经置换后 PC II 送出 48 比特，作为第 $i$ 次迭代时所用的子密钥 $K_i$。子密钥发生器如图 1.1.3 所示。

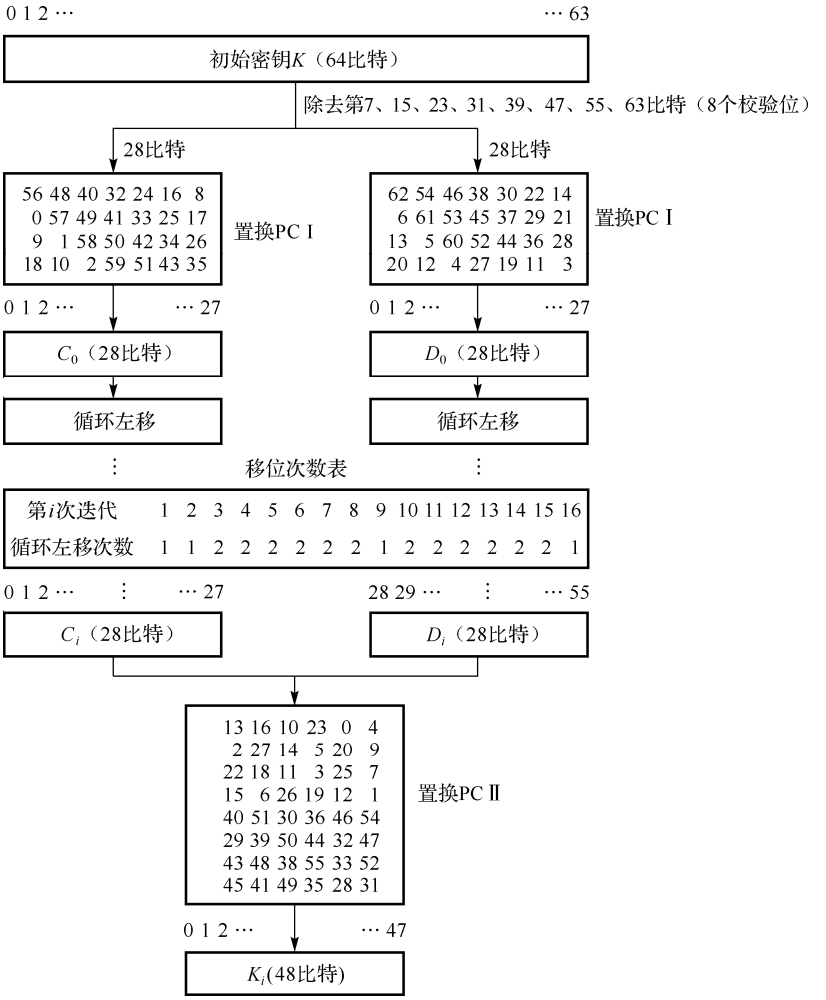



图 1.1.3　子密钥发生器

DES 加密算法可以简单归结如下。

加密过程：

$$L_0,R_0 \leftarrow \text{IP（64 比特输入）}$$

$$L_i \leftarrow R_{i-1}$$

$$R_i \leftarrow L_{i-1} \oplus f(L_i,\cdots,K_i)，\ i=1,2,\cdots,16$$

$$\text{（64 比特密文）} \leftarrow \text{IP}^{-1}(R_{16}L_{16})$$

DES 的加密运算是可逆的，其解密过程与加密过程类似。

解密过程：

$$R_{16},L_{16} \leftarrow \text{IP（<64 比特密文>）}$$

$$R_{i-1} \leftarrow L_i$$

$$L_{i-1} \leftarrow R_i \oplus f(R_{i-1},K_i)，\ i=16,15,\cdots,1$$

$$\text{<64 比特明文>} \leftarrow \text{IP}^{-1}(L_0R_0)$$

这里给出了一个加密过程的例子（用十六进制表示）：

| | |
|---|---|
| 密钥： | 03 96 48 C5 39 31 39 65 |
| 明文： | 00 00 00 00 00 00 00 00 |
| 经置换 IPRG： | 00 00 00 00 00 00 00 00 |
| 第 1 层： | 00 00 00 00 85 7E 2A 43 |
| 第 2 层： | 85 7E 2A 43 D7 2F 0D 7B |
| 第 3 层： | D7 2F 0D 7B C7 6E 6C B1 |
| 第 4 层： | C7 6E 6C B1 4C B0 77 8A |
| 第 5 层： | 4C B0 77 8A 72 2B BC 81 |
| 第 6 层： | 72 2B BC 81 59 85 72 7B |
| 第 7 层： | 59 85 72 7B 82 67 AE 9C |
| 第 8 层： | 82 67 AE 9C E7 DD DB 94 |
| 第 9 层： | E7 DD DB 94 71 90 0F 11 |
| 第 10 层： | 71 90 0F 11 0A AD 33 E4 |
| 第 11 层： | 0A AD 33 E4 51 61 B2 81 |
| 第 12 层： | 51 61 B2 81 7D DD 4A 9E |
| 第 13 层： | 7D DD 4A 9E 75 17 39 28 |
| 第 14 层： | 75 17 39 28 9D A0 1E 4E |
| 第 15 层： | 9D A0 1E 4E BB 14 FC F2 |
| 第 16 层： | 73 6A 7F 8A BB 14 FC F2 |
| 经过 $IP^{-1}$ 后的密文： | C4 D7 2C 9D EE DE 5E 8B |

### 1.1.3　DES 的工作方式

DES 的工作方式也适用于其他分组密码。

（1）电子密本（ECB）方式。将明文<$X$>分成 $m$ 个 64 比特分组，即

$$<X>=(x_0,x_1,\cdots,x_{m-1})$$

如果明文长度不是 64 比特的整数倍，则在明文末尾填补适当数目的规定符号。ECB 方式对明文分组用给定的密钥 $K$ 分别进行加密，可得密文

$$<Y> = (y_0,y_1,\cdots,y_{m-1})$$

式中，$y_i$=DES($K$,$x_i$)，$i$=0,1,⋯,$m$−1。这种工作方式组间同明即同密，组间可能出现重复。

（2）密文分组连接（CBC）方式。在 CBC 方式下，在对每个明文分组 $x_i$ 加密之前先与前一分组的密文 $y_{i-1}$ 按位进行模 2 加后，再进行 DES 加密，需预置一个初始向量(**IV**)$y_{-1}$=**IV**。

CBC 方式：　　$<X> \rightarrow <Y>, <Y> = \mathrm{CBC}(k,<X>)$

$$y_{-1} = \mathbf{IV}$$

$$y_j = \mathrm{DES}(K,x_j \oplus y_{j-1}), \qquad 0 \leqslant j \leqslant m$$

脱密 $\mathrm{CBC}^{-1}$：　　$<Y> \rightarrow <X>$

$$y_{-1} = \mathbf{IV}$$

$$x_j = \mathrm{DES}^{-1}(K,y_j) \oplus y_{j-1}$$

CBC 方式克服了 ECB 方式组间重复的缺点，但由于明文分组的加密与前一分组密文有关，因此前一分组密文的错误会传播到下一分组。

（3）密文反馈（CFB）方式（可用于序列密码）。设明文为<$X$>=($x_0$,$x_1$,⋯,$x_{m-1}$)，其中 $x_i$ 由 $t$ 比特组成，0<$t$⩽64。

由图 1.1.4 可知，CFB 方式实际上将 DES 作为一个密钥流发生器，在 $t$ 比特密文的反馈下，每次输出 $t$ 比特乱数来对 $t$ 比特明文进行加密，即：

$$<X> \rightarrow <Y>$$

$$y_i = \mathrm{left}_t[\mathrm{DES}(k,Z_i)] \oplus x_i$$

或

$$y_i = \mathrm{right}_t[\mathrm{DES}(k,Z_i)] \oplus x_i$$

式中，$Z_0$ = **IV**（初始向量为 64 比特）。

$$Z_i = \mathrm{right}_{64}[Z_{i-1} \| y_{i-1}], \qquad 1 \leqslant i < m-1$$

式中，“‖”表示相接，$\mathrm{left}_t[\cdot]$表示取左边的 $t$ 比特，$\mathrm{right}_t[\cdot]$表示取右边的 $t$ 比特。

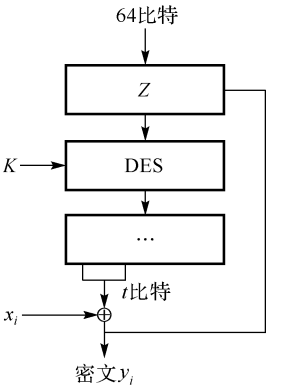



图 1.1.4　CFB 方式示意图

由于 CFB 方式采用密文反馈，因此对密文错误（通常由信道传输等造成）比较敏感，$t$ 比特密文中只要有 1 比特的错误，就会导致连续数个 $t$ 比特出错。

（4）输出反馈（OFB）方式（可用于序列密码）。

由图 1.1.5 可看出，OFB 方式与 CFB 方式的不同点只是 OFB 方式直接取 DES 加密算法输出的 $t$ 比特作为反馈（而 CBF 方式以密文 $y_i$ 作为反馈），其余都与 CFB 方式相同。

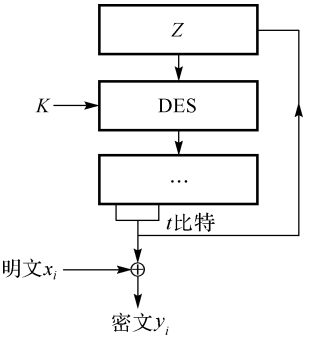



图 1.1.5　OFB 方式示意图

$$<X> \rightarrow <Y>$$

$$y_i = \text{left}_t[\text{DES}(k, Z_i)] \oplus x_i$$

或

$$y_i = \text{right}_t[\text{DES}(k, Z_i)] \oplus x_i$$

$$Z_0 = \mathbf{IV}\text{（初始向量为 64 比特）}$$

$$Z_i = \text{right}_{64}\{Z_{i-1} \parallel \text{left}_t[\text{DES}(k, Z_{i-1})]\}$$

或

$$Z_i = \text{right}_{64}[Z_{i-1} \| \text{right}_t[\text{DES}(k, Z_{i-1})]]$$

OFB 方式以 DES 加密算法的输出作为反馈，因而克服了 CFB 方式密文错误传递的缺点。

### 1.1.4　DES 加密思想和特点

香农（Shannon）曾建议使用不同类型的函数（如反复、交替地进行代替和简单的线性变换）来构成一个混搅变换（Mixing Transformation），通过这种变换可以把明文消息随机、均匀地分布在所有可能的密文消息集合上。DES 加密算法每层的 $f$ 函数包含加乱（$\oplus K_i$）、代替（$S$ 盒）、移位（置换 $P$），也就是反复、交替地使用这些变换使输出的每个比特都依赖于整个输入组，使加密的每个比特都依赖于整个密钥，即通过多层的反复变换，使密文的每个比特都是明文和密钥的完全函数。

DES 加密算法采用扩张函数 $E$ 实现 $S$ 盒和 $P$ 置换，采用 $S$ 盒进行小块的非线性变换，以达到混合（Confusion）；采用 $P$ 置换进行大块的线性变换，以达到扩散（Diffusion）。因此，$S$ 盒的设计是 DES 加密算法的一个核心问题。实际使用的 $S$ 盒是经过精心设计和严格挑选的，因为 $S$ 盒的某些设计原则是敏感的，NSA 给出了下列三条属于设计准则。

① $S$ 盒的输出不是输入的线性函数或仿射函数。

② 改变 $S$ 盒的 1 个输入比特，就至少可引起 $S$ 盒的 2 个输出比特不同。

③ $S(X)$与 $S(X+001100)$至少有 2 个比特不同。

DES 加密算法具有互补性的特点，假设对明文 $X$ 逐位取补，记为 $\overline{X}$；密钥逐位取补，记为 $\overline{K}$。若密文组为：

$$Y = \text{DES}_K(X)$$

则有

$$\overline{Y} = \text{DES}_K^{-1}(\overline{X})$$

式中，$\overline{Y}$ 是 $Y$ 的逐位取补。互补性特点是由 DES 加密算法中的两次异或运算决定的，一次是在 $S$ 盒之前，另一次是在 $P$ 置换之后。

DES 加密算法存在弱密钥和半弱密钥。

DES 加密算法每次迭代都有一个子密钥供加密使用，16 次迭代子密钥分别为 $K_1, K_2, \cdots, K_{16}$，由初始密钥 $K$ 生成。如果 $K$ 通过子密钥发生器生成的 $K_1, K_2, \cdots, K_{16}$ 都相同，则称密钥 $K$ 为弱密钥，DES 加密算法存在 4 个弱密钥。如果用初始密钥 $K$ 对明文 $X$ 进行 2 次加密或 2 次解密，都可恢复出明文（即加密运算与解密运算没有区别），则称这类密钥 $K$ 为半弱密钥，即

$$\text{DES}_K[\text{DES}_K(X)] = X$$

$$\text{DES}_K^{-1}[\text{DES}_K^{-1}(X)] = X$$

DES 加密算法存在 6 对半弱密钥。

弱密钥和半弱密钥是不安全的，属于禁用密钥。当采用随机途径生成密钥时，要检查并删除禁用密钥。

## 1.2 Lucifer 加密算法（DES 前身）

Lucifer 加密算法是美国 IBM 公司 Watson 实验室在 1970 年研制的用于数据加密的分组密码体制。

Lucifer 加密算法的输入分组、输出分组和密钥分组均为 128 比特（16 字节）。Lucifer 是乘积密码，加密变换由 16 层（Round）变换迭代而成，每层都对输入进行 C-I-D 变换：C 代表混乱（Confusion），I 代表密钥加扰（Key Interruption），D 代表扩散（Diffusion）。

输入消息分组（明文分组）的 16 字节（128 比特）分成左半组和右半组，层与层之间的左、右半组进行变换，如图 1.2.1 所示。

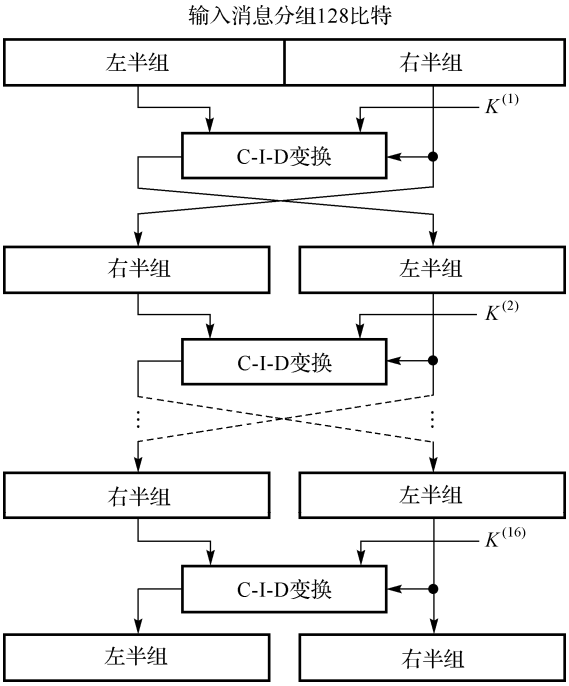



图 1.2.1　Lucifer 加密算法

在每一层中，输入的 8 字节要经过 C-I-D 变换，如图 1.2.2 所示。

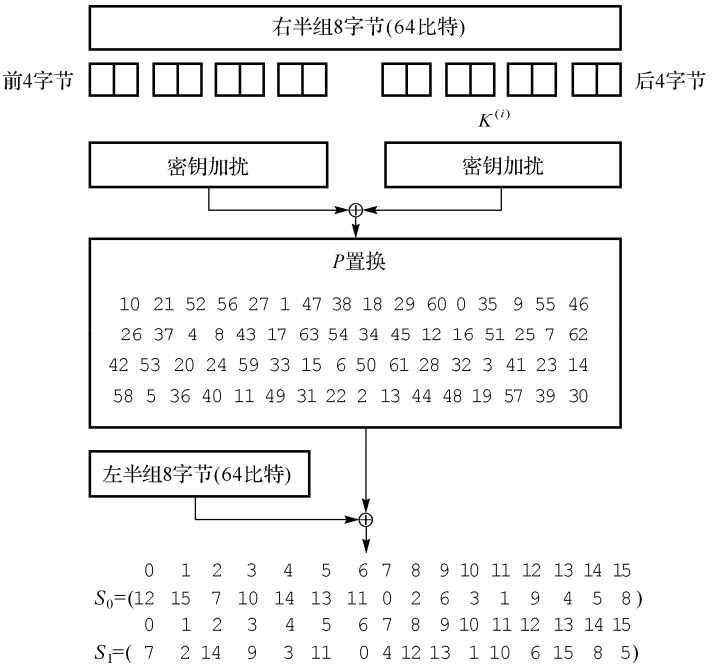



图 1.2.2　C-I-D 变换

（1）经过 $S_0$ 或 $S_1$ 代替。8 个消息字节 $M_0,M_1,\cdots,M_7$，每个字节的左 4 比特与右 4 比特分别进入 $S_1$ 或 $S_0$ 进行代替，在第 $i$ 层，由层密钥 $K^{(i)}$（$1\leqslant i\leqslant 16$）的第一个字节的 8 个比特（密钥比特）分别控制 8 个消息字节选择 $S_0$ 和 $S_1$。当密钥比特为 0 时，对应的消息字节的左 4 比特进入 $S_1$，右 4 比特进入 $S_0$；当该密钥比特为 1 时，对应的消息字节的左 4 比特进入 $S_0$，右 4 比特进入 $S_1$。第 1 层迭代时，层密钥 $K^{(i)}=(K_0^{(1)},K_1^{(1)},\cdots,K_7^{(1)})$，控制字节 $K_0^{(1)}$ 的第 0,1,2,…,7 比特分别控制 $M_7,M_6,\cdots,M_0$，以选择 $S_0$ 或 $S_1$。在第 $i$ 层代替时，层密钥 $K^{(i)}$如表 1.2.1 所示。控制字节 $K_0^{(i)}$ 为表中第 1 列所指的密钥字节。

$S_0$、$S_1$ 是输入为 4 比特、输出为 4 比特的非线性变换，把输入、输出的 4 比特都看成 0～15 的数，$S_0$、$S_1$ 可看成 0,1,…,15 到 0,1,…,15 的一个 16 元置换。

（2）与密钥字节进行 xor 运算。经过 $S_0$ 和 $S_1$ 代替后的 8 字节与指定的密钥字节进行 xor 运算（对应比特分别进行模 2 加）。

（3）经 $P$ 置换后调整比特位置。经过与密钥字节进行 xor 运算后的 64 比特进行图 1.2.2 中的 $P$ 置换，即新的 64 比特的第 0 比特是原来的第 10 比特，第 1 比特是原来的第 21 比特，…，第 63 比特是原来的第 30 比特。

（4）与左半组（64 比特）分别进行 xor 运算。经 $P$ 置换后的 64 比特分别与左半组的 64 比特进行模 2 加，得到的新的 64 比特为 C-I-D 变换的输出。

表 1.2.1　层密钥

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ←消息字节 |
|---|---|---|---|---|---|---|---|---|---|
| 第 1 层 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 第 2 层 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| 第 3 层 | 14 | 15 | 0 | 1 | 2 | 3 | 4 | 5 | |
| 第 4 层 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| 第 5 层 | 12 | 13 | 14 | 15 | 0 | 1 | 2 | 3 | |
| 第 6 层 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 第 7 层 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | |
| C-I-D→ 第 8 层 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
| 第 9 层 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | |
| 第 10 层 | 15 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
| 第 11 层 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |
| 第 12 层 | 13 | 14 | 15 | 16 | 0 | 1 | 2 | 3 | |
| 第 13 层 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
| 第 14 层 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 | |
| 第 15 层 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| 第 16 层 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | |

脱（解）密实际上是将密码分组（128 比特）作为输入，从第 16 层到第 1 层进行逆变换，每层的左半组与右半组交换位置，C-I-D 变换也倒过来，按密钥从 $K^{16}, K^{15},\cdots, K^{1}$，从第 16 层到第 1 层，密文分组从第 16 层进行到第 1 层，即可得到明文分组。

作为 DES 加密算法的前身，Lucifer 加密算法与 DES 加密算法有许多相似的地方。在加密时，明文分组都要经过 16 层变换，明文分组分成左右两个半组交替进行变换，并相互渗透。每层的变换都通过 *S* 盒来进行混乱，通过与密钥字节进行模 2 加来实现加扰，通过与另一个半组进行模 2 加来实现扩散。

Lucifer 加密算法通过左右两个半组相互交替进行多层混乱-加扰-扩散，从而实现对明文分组的加密，使得密文分组的每个比特与明文分组的每个比特都有关，尽管破译者得到密文也不易分析。与 DES 加密算法比较，Lucifer 加密算法中每层的变换比 DES 加密算法简单得多。例如，*S* 盒只有 2 个，每个只含一个置换，没有扩张变换。另外，DES 加密算法先与层密钥加乱，再经 *S* 盒代替；而 Lucifer 加密算法先经 *S* 盒代替，再与层密钥加乱。Lucifer 加密算法的输入、输出和密钥分组长均为 128 比特，比 DES 加密算法长得多，能有效地抵抗穷举破译法。

## 1.3　NewDES 加密算法

NewDES 加密算法是 Robert Scott 于 1985 年设计的，其目的是成为 DES 加密算法的替代

算法。NewDES 加密算法是基于 64 比特的明文分组进行运算的，密钥长度为 120 比特。在设计上，所有的运算都针对完整的字节，而不像 DES 加密算法那样要进行大量特定比特的读写或置换操作，这使得 NewDES 加密算法的软件实现更为简单方便。

### 1.3.1　NewDES 加/脱密编制

将 64 比特的明文分组分成 8 个单字节长的子分组 $B_0$、$B_1$、…、$B_6$、$B_7$，这些子分组要运行 17 轮，每一轮有 8 步。在每一步中，其中一个子分组与某个密钥进行模 2 加，然后用非线性函数 $f$ 输出代替字节，接着与另一个子分组进行模 2 加并生成这个子分组。120 比特的密钥分成 15 个密钥子分组 $K_0$、$K_1$、…、$K_{13}$、$K_{14}$。NewDES 加密算法如图 1.3.1 所示。

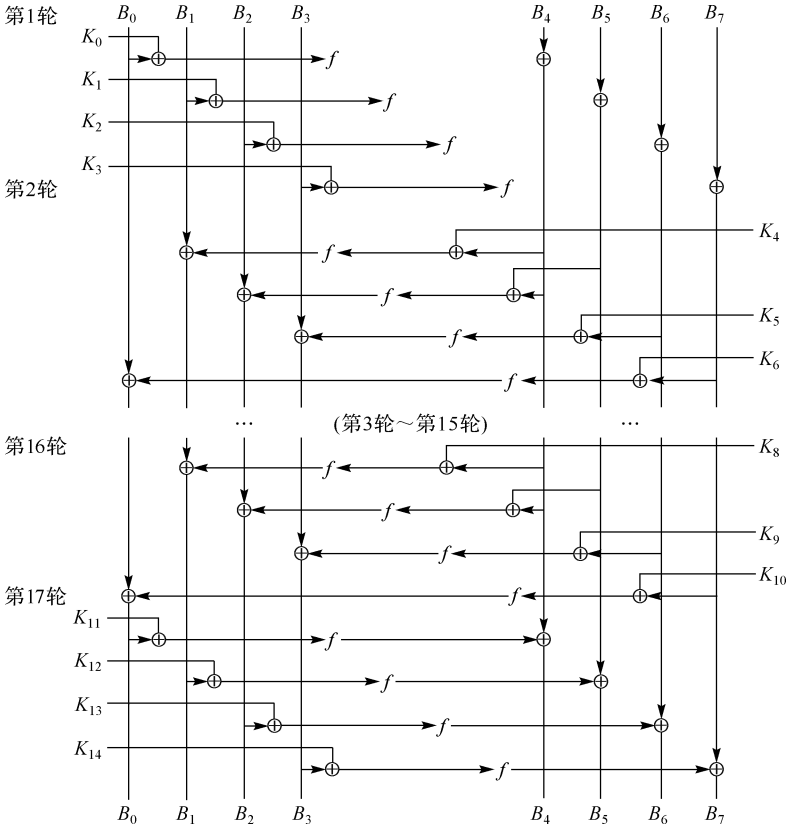



图 1.3.1　NewDES 加密算法

NewDES 加/脱密算法的 C 语言表述如下：

```
char B0,B1,B2,B3,B4,B5,B6,B7;
char Key[15];
int initi,delta,fini;
```

当加密时，初始设置为：

```
initi=0;
delta=1;
fini=0;
```

当脱密时，初始设置为：

```
initi=11;
delta=9;
fini=12;
```

相关代码如下：

```
NewDES()
{   int  i;
    i=initi;
    while(1)
    {   B4=B4^f[B0^Key[i++]];
        if(i==15)
           i=0;
        B5=B5^f[B1^Key[i++]];
        if(i==15)
           i=0;
        B6=B4^f[B2^Key[i++]];
        if(i==15)
           i=0;
        B7=B7^f[B3^Key[i]];
        i=i+delta;
        if(i>14)
           i=i-15;
        if(i==fini)
           return;
        B1=B1^f[B4^Key[i++]];
        B2=B2^f[B4^B5];
        B3=B3^f[B6^Key[i++]];
        B4=B0^f[B7^Key[i]];
        i=i+delta;
        if(i>14)
           i=i-15;
    }
}
```

### 1.3.2　非线性函数 *f* 的选取

NewDES 加/脱密编制中的非线性函数 *f* 实际上是一个字节到字节的伪随机置换，其输入和输出都是 0～255 之间的一个数。

Robert Scott 在设计此函数时以著名的美国“独立宣言”为输入文本，通过执行如下过程：

```
for(i=0; i<256; i++)
f[i]=i;
i=0; j=0;
while( (c=getchar())!=EOF )
{
    if(c>'Z')
    c=c-32;
    if(c>='A'&& c<='Z')
    {
        i=(i+1) & 255;
        j=(j+c) & 255;
        k=f[i];
        f[i]=f[j];
        f[j]=k;
    }
}
```

得到的置换表为

```
f[0……255]=
 23  137  239  188  102  125  221   72  212   68   81   37   86  237  147  149
 70  229   17  124  115  207   33   20  122  143   25  215   51  183  138  142
146  211  110  173    1  228  189   14  103   78  162   36  253  167  116  255
158   45  185   50   98  168  250  235   54  141  195  247  240   63  148    2
224  169  214  180   62   22  117  108   19  172  161  159  160   47   43  171
194  175  178   56  196  112   23  220   89   21  164  130  157    8   85  251
216   44   94  179  226   38   90  119   40  202   34  206   35   69  231  246
 29  109   74   71  176    6   60  145   65   13   77  151   12  127   95  199
 57  101    5  232  150  210  129   24  181   10  121  187   48  193  139  252
219   64   88  233   96  128   80   53  191  144  218   11  106  132  155  104
 91  136   31   42  243   66  126  135   30   26   87  186  182  154  242  123
 82  166  208   39  152  190  113  205  114  105  225   84   73  163   99  111
204   61  200  217  170   15  198   28  192  254  134  234  222    7  236  248
201   41  177  156   92  131   67  249  245  184  203    9  241    0   27   46
133  174   75   18   93  209  100  120   76  213   16   83    4  107  140   52
 58   55    3  244   97  197  238  227  118   49   79  230  223  165  153   59
```

## 1.4 DES 加密算法的变型

### 1.4.1 多重 DES 加密算法

通过多次使用 DES 加密算法就可以获得更强的密码，密钥长度也可以成倍地增长，但其代价是速度会减慢。在多重 DES 加密算法中，以密钥长度为 112 比特的三重 DES 加密算法最为常见，其框图如图 1.4.1 所示。

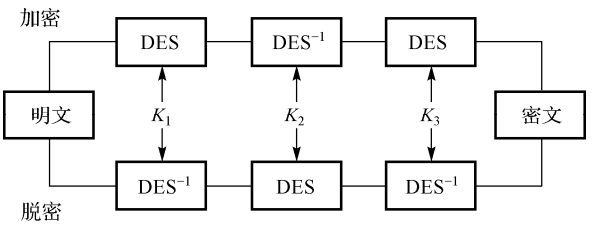



图 1.4.1 三重 DES 加密算法框图

### 1.4.2 子密钥独立的 DES 加密算法

在子密钥独立的 DES 加密算法中，每轮的子密钥都是独立的，不是由单个的 56 比特密钥生成的。因为 16 轮中的每轮都要求 48 比特的密钥，这就意味着这种变型的密钥长度为 768 比特，极大地增加了 DES 加密算法的实现难度。

比哈姆（Eli Biham）和沙米尔（Adi Shamir）证明了该变型仍可用 $2^{61}$ 个选择明文来破译，并不需要强有力地攻击 $2^{768}$ 次，因此，改变密钥的生成方式并不能使 DES 加密算法变得更安全。

### 1.4.3 *S* 盒可选的 DES 加密算法

*S* 盒可选的 DES 加密算法是指将 DES 加密算法的改变集中在 *S* 盒上。在一些设计中，*S* 盒的次序是随密钥变化的，也有一些设计者使 *S* 盒的内容是可变的。但比哈姆和沙米尔证明，DES 加密算法的 *S* 盒的设计，甚至 *S* 盒自身的内容抗差分分析都是最优的。DES 加密算法的 8 个 *S* 盒次序的改变（并不改变 *S* 盒的内容）都会使得 DES 加密算法变弱许多，甚至 DES 加密算法的一个 *S* 盒的内容稍做改变，也会导致 DES 内容易于破译。

目前，还没有人证明，就所知的攻击方法而言，会有更好的 *S* 盒。

### 1.4.4 *S* 盒可变的 DES 加密算法

差分分析仅在分析者已知 *S* 盒结构的前提下才有效，一些设计者使 *S* 盒根据密钥进行变化，并使其保密，那么差分分析就不再适用了。

### 1.4.5　CRYPT(3)加密算法

CRYPT(3)加密算法是 UNIX 操作系统上 DES 的加密算法变型，主要用于对口令实施单向变换，也可用于加密，是一个 $S$ 盒和 $E$ 盒均与密钥相关的 DES 加密算法的变型。

### 1.4.6　广义的 DES（GDES）加密算法

GDES 加密算法是既为了提高 DES 加密算法的速度，又为了提高其强度而设计的，该算法将明文分组长度增加，非线性函数 $f$ 则保持不变。GDES 加密算法对不同长度的明文分组进行操作，将加密分组分成 $q$ 个 32 比特的子分组，精确数目依赖于整个分组长度（在设计中分组长度是可变的，但在实现中却必须固定），通常令 $q$ 等于明文分组长度除以 32。GDES 加密算法编制框图如图 1.4.2 所示。

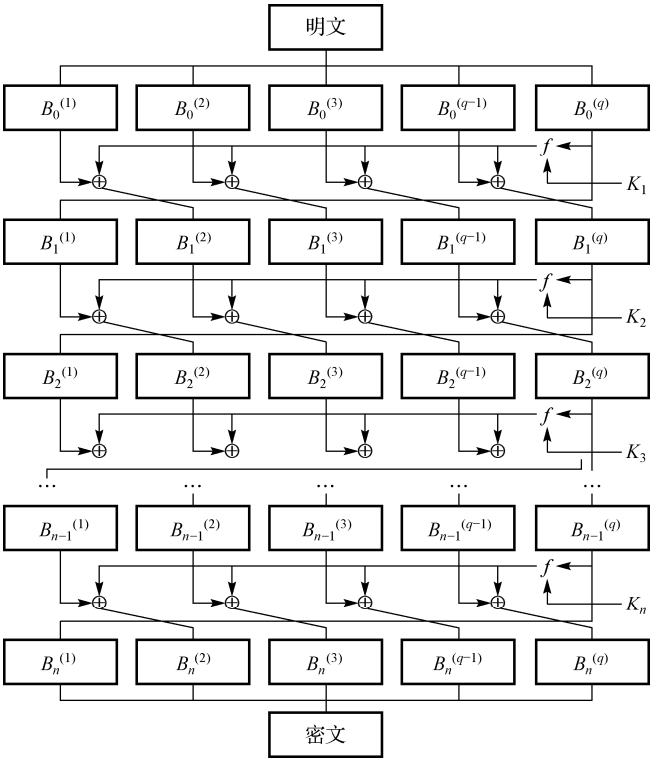



图 1.4.2　GDES 加密算法编制框图

非线性函数 $f$ 在每轮最右边的分组上计算一次，其结果同其他部分进行模 2 加，然后将模 2 加后的结果循环右移。GDES 加密算法的轮数 $n$ 是不变的，最后一轮稍有不同，使得加密和脱密过程仅在子密钥的次序上有所不同（如 DES 加密算法）。事实上，当 $q$=2、$n$=16 时，GDES 加密算法就是 DES 加密算法。

比哈姆和沙米尔证明了，使用差分分析破译 $q$=8、$n$=16 的 GDES 加密算法仅需 6 个选择明文。如果使用不相关的子密钥，则仅需 16 个已知明文。$q$=8、$n$=22 的 GDES 加密算法可以由 48 个选择明文破译；$q$=8、$n$=31 的 GDES 加密算法仅要 50 万个选择明文就能破译；$q$=8、$n$=64 的 GDES 加密算法需要 $2^{49}$ 个已知明文方能破译，仍然比 DES 加密算法更弱。事实上，比 DES 加密算法快的任何 GDES 加密算法都不如 DES 加密算法安全。DES 加密算法变型的差分分析如表 1.4.1 所示。

表 1.4.1　DES 加密算法变型的差分分析

| 变 型 运 算 | | 选 择 明 文 |
|---|---|---|
| 完整 DES 加密算法（无变型） | | $2^{47}$ |
| $P$ 置换 | | 不能变强 |
| 恒等置换 | | $2^{19}$ |
| 改变 $S$ 盒顺序 | | $2^{38}$ |
| 用模 2 加 | | $2^{38}$,$2^{31}$ |
| $S$ 盒 | 随机 | $2^{21}$ |
| | 随机置换 | $2^{44}$～$2^{48}$ |
| | 改变一项 | $2^{33}$ |
| | 均匀表 | $2^{26}$ |
| 去掉 $E$ 扩散 | | $2^{26}$ |
| 改变 $E$ 和子密钥模 2 加的顺序 | | $2^{44}$ |
| 广义 DES（$q$=8） | 16 轮 | 6,16 |
| | 64 轮 | $2^{49}$（独立的密钥） |

### 1.4.7　DESX 加密算法

DESX 是 RSA 数据安全公司设计的 DES 加密算法的一种变型。自 1986 年以来，它就一直是 MailSafe 电子邮件安全程序的密码算法，同时它也是 BSafe 工具箱中的密码算法之一。DESX 加密算法在加/脱密时仍然使用 DES 加密算法，只是在 DES 加密算法的输入和输出中添加了附加密钥。具体情形是，除了 56 比特的 DES 加密算法密钥，DESX 加密算法还有一个 64 比特的附加密钥，在 DES 加密算法的第一轮之前，64 比特的附加密钥同明文进行模 2 加，随后使用一个单向函数，以整个 120 比特的密钥作为输入进行计算后得到一个 64 比特的生成密钥，并在 DES 加密算法的最后一轮与密文进行模 2 加。这种方式的变型使 DESX 加密算法比 DES 加密算法有更强的抵抗穷尽搜索攻击能力，同时也改进了抗差分分析和线性分析的能力，它分别要求 $2^{61}$ 个选择明文和 $2^{60}$ 个已知明文。

### 1.4.8　Ladder-DES 加密算法

Ladder-DES 加密算法将 DES 加密算法作为每轮使用的非线性函数，其框图如图 1.4.3 所示。

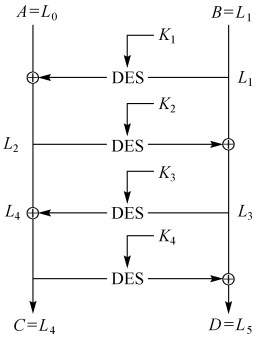



图 1.4.3　Ladder-DES 加密算法框图

## 1.5　FEAL 加密算法

FEAL 是由日本 NTT 电子通信实验室的 A.Schimizu 和 S.Miyaguchi 于 1987 年提出的一个分组加密算法，采用平衡 Feistel 结构，几乎同 DES 加密算法完全一样。它的分组长度为 64 比特，迭代层数为 $N$（$N$=2$^x$，$x$>1）。密钥长度有 128 比特和 64 比特两种类型，前者称为 FEAL-NX，后者称为 FEAL-N，当前者的后 64 比特全为 0 时，两者就完全一致了。FEAL 加密算法以字节为单位进行运算，便于软件快速实现。

FEAL 加密算法框图如图 1.5.1 所示。

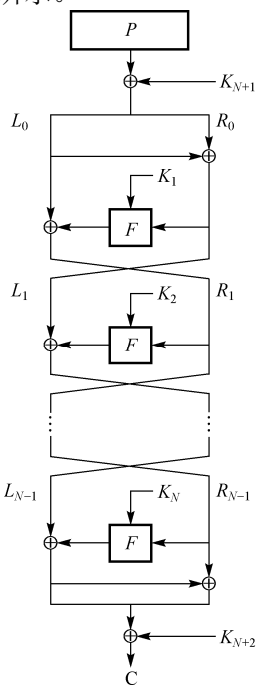



图 1.5.1　FEAL 加密算法框图

明文 $P$ 同子密钥 $K_{N+1}$ 进行模 2 加后被平分成 32 比特的左右两部分 $L_0$ 和 $R_0$，$L_0$ 和 $L_0 \oplus R_0$ 作为第一层的输入，经过 $N$ 层的迭代过程如下：

$$L_i=R_{i-1}$$

$$R_i=L_{i-1} \oplus F(R_{i-1},K_i), \qquad i=1,2,\cdots,N$$

对（$L_N$,$R_N$）进行如下变换：

$$(L_N,R_N)=(R_N,L_N)$$

$$R_N=R_N \oplus L_N$$

$$(L_N,R_N)=(L_N,R_N) \oplus K_{N+2}$$

（$L_N$,$R_N$）即所求的密文。层密钥 $K_1,K_2,\cdots,K_N$ 的长度为 2 字节，$K_{N+1}$、$K_{N+2}$ 的长度为 8 字节，层密钥的生成规律见 1.5.2 节。

### 1.5.1 层函数 *F*

层函数 $F$ 对 2 字节的层密钥和 4 字节的输入进行模 2 加、模 256 加和循环移位操作后，得到 4 字节的输出。记 2 字节的层密钥为（$\beta_0$,$\beta_1$），4 字节的输入为（$\alpha_0$,$\alpha_1$,$\alpha_2$,$\alpha_3$），则层函数 $F$ 的 4 字节输出（$c_0$,$c_1$,$c_2$,$c_3$）为：

$$Z_1=\alpha_1 \oplus \beta_0 \oplus \alpha_0$$

$$Z_2=\alpha_2 \oplus \beta_1 \oplus \alpha_3$$

$$c_0=S_0(\alpha_0,c_1)$$

$$c_1=S_1(Z_1,Z_2)$$

$$c_2=S_0(Z_2,c_1)$$

$$c_3=S_1(\alpha_3,c_2)$$

式中，$S_0$ 和 $S_1$ 为两个 $S$ 盒，它们的输入为 2 字节，输出为 1 字节。这两个 $S$ 盒分别由下面两个公式生成（其中“<<<”表示循环左移位）：

$$S_0(X_1,X_2) = [(X_1+X_2)\bmod 256] <<< 2$$

$$S_1(X_1,X_2) = [(X_1+X_2+1)\bmod 256] <<< 2$$

在程序实现中可以把整个层函数 $F$ 看成一个 48 比特输入、32 比特输出的非线性 $S$ 盒。

### 1.5.2 层密钥生成规律

先介绍密钥 $K$ 为 128 比特的情形。由 128 比特的密钥 $K$ 生成 $K_{N+2}$ 个层密钥是通过($N$/2+4)层的迭代过程来完成的，如图 1.5.2 所示。

将输入密钥 $K$ 平分成 64 比特的左右两部分 $K_L$ 和 $K_R$，又将 $K_L$ 平分成 32 比特的左右两部分 $A_0$ 和 $B_0$，将 $K_R$ 平分成 32 比特的左右两部分 $K_{R1}$ 和 $K_{R2}$，经过($N$/2+4)次迭代过程（其中“>>>”表示循环右移位）：

$$A_r=B_{r-1}$$

$$B_r=f(A_{r-1},\ B_{r-1}\oplus Q_r\oplus A_{r-2})$$

$$K_{2r-1}=B_r >>> 16$$

$$K_{2r}=B_r\&0\text{xffff},\qquad r=1,2,\cdots,N/2+4$$

$$K_{N+1}=K_{N+1}\parallel K_{N+2}\parallel K_{N+3}\parallel K_{N+4}$$

$$K_{N+2}=K_{N+5}\parallel K_{N+6}\parallel K_{N+7}\parallel K_{N+8}$$

当 $r$=1 时，定义 $A_{r-1}$=0。

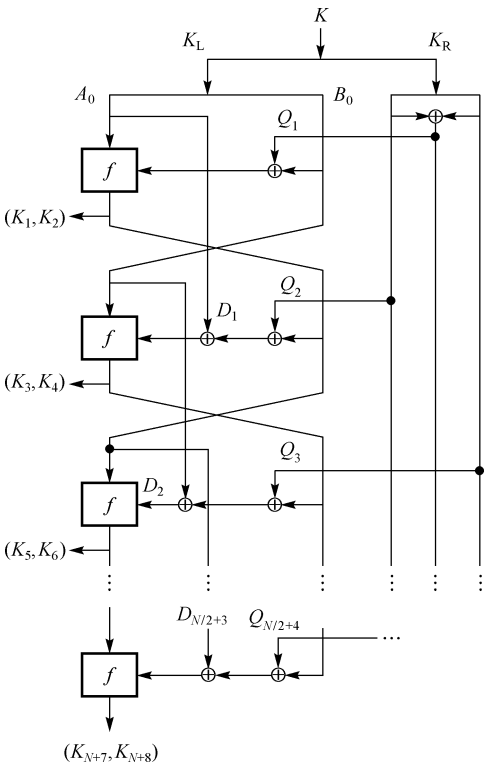



图 1.5.2　FEAL 加密算法的层密钥生成框图

$Q_r$ 由 $K_{\mathrm{R}}$ 来决定，即

$$Q_r=\begin{cases}K_{\mathrm{R2}}, & r=3i\\ K_{\mathrm{R1}}\oplus K_{\mathrm{R2}}, & r=3i+1\\ K_{\mathrm{R1}}, & r=3i+2\end{cases}\qquad i\in Z$$

$f$ 函数（DES 加密算法的核心）和层函数 $F$（层函数）非常相似，它的输入为 8 字节，输出为 4 字节。记输入的 8 字节为（$\alpha_0,\alpha_1,\alpha_2,\alpha_3,\beta_0,\beta_1,\beta_2,\beta_3$），输出的 4 字节为（$f_0,f_1,f_2,f_3$），则 $f$ 函数为：

$$f_1 = \alpha_1 \oplus \alpha_0$$
$$f_2 = \alpha_2 \oplus \alpha_3$$
$$f_1 = S_1(f_1,\ f_2 \oplus \beta_0)$$
$$f_2 = S_0(f_2,\ f_1 \oplus \beta_1)$$
$$f_0 = S_0(\alpha_0,\ f_1 \oplus \beta_2)$$
$$f_3 = S_1(\alpha_3,\ f_2 \oplus \beta_3)$$

$S_0$ 和 $S_1$ 同层函数 $F$ 中的 $S_0$ 和 $S_1$。

当输入密钥 $K$ 为 64 比特时，在尾部补 64 个 0，即可用上面的方法来生成层密钥，此时 $K_R$ 全为 0，$Q_r$ 也全为 0。

FEAL 加密算法的脱密过程和加密过程一致。

## 1.6 LOKI 加密算法

LOKI 是由澳大利亚的 L.Brown 和 J.Seberry 等人提出的一种 16 层 DES 型迭代分组加密算法，该加密算法于 1989 年提出，又于 1991 年做了修改，原方案称为 LOKI89，修改后的方案称为 LOKI91。LOKI 加密算法和 DES 加密算法非常相似，也是由 $E$ 扩展、$S$ 盒代替、$P$ 置换，以及与层密钥进行模 2 加来构建层函数的。LOKI 加密算法的分组长度为 64 比特，密钥长度也为 64 比特，迭代层数为 16。LOKI91 加密算法是 AES 的候选算法之一，采用 16 层的平衡 Feistel 结构，其分组长度为 128 比特，密钥可以为 128、192、256 比特。

### 1.6.1 LOKI89 加密算法

明文 $P$ 首先与密钥进行模 2 加，再被分成 32 比特的左右两部分，然后经过如下所示的 16 层迭代变换。

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1} \oplus K_{i-1}), \qquad i = 1,2,\cdots,16$$

最后一层不变换，第 16 层输出与层密钥 $K_{16}$ 进行模 2 加后即可得到密文。

LOKI91 加密算法去掉了 LOKI89 加密算法中的输入、输出与密钥的结合过程。

$F$ 函数由 $E$ 扩展、$S$ 盒代替和 $P$ 置换三部分组成。

$E$ 扩展按表 1.6.1 所示的 $E$ 扩展表将 32 比特扩展成 48 比特。

表 1.6.1 $E$ 扩展表

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 0 | 1 | 2 | 3 |

LOKI 加密算法使用了 4 个相同的 $S$ 盒。$S$ 盒的输入为 12 比特，输出为 8 比特。LOKI 加密算法首次使用了有限域上的多项式来构建 $S$ 盒，它使用了 16 个 $F_2$ 上的 8 次不可约多项式，通过有限域 GF($2^8$)中的模幂运算来实现 $S$ 盒代替。我们把 $F_2$ 上的多项式简记为由它们系数组成的二元数组，例如，将多项式 $x^8+x^6+x^5+x^4+x^2+x+1$ 简记为（101110111），$S$ 盒的计算使用下列 16 个 $F_2$ 上的 8 次不可约多项式，即

| $r$ | $g_r$ |
|---|---|
| 0 | 101110111 |
| 1 | 101111011 |
| 2 | 110000111 |
| 3 | 110001011 |
| 4 | 110001101 |
| 5 | 110011111 |
| 6 | 110100011 |
| 7 | 110101001 |
| 8 | 110110001 |
| 9 | 110111101 |
| 10 | 111000011 |
| 11 | 111001111 |
| 12 | 111010111 |
| 13 | 111011101 |
| 14 | 111100111 |
| 15 | 111110011 |

二元数组与 $F_2$ 上的多项式相对应，在 $S$ 盒的计算中需要相互转换。

将输入的 48 比特从左到右每 12 比特输入一个 $S$ 盒，将 12 比特记为 $b_0\cdots b_{11}$，令 $r$ 为（$b_2b_3b_4b_5b_6b_7b_8b_9$）的二进制值，$c$ 为（$b_0b_1b_{10}b_{11}$）的二进制值，则 $S$ 盒的输出值为

$$o=(c+r)^{31} \bmod g_r$$

在 LOKI91 加密算法中，$S$ 盒的输出值为

$$f=c+[(r\times 17)\oplus 0\text{xff}] \bmod 256$$

将 $f$ 看成 $F_2$ 上的多项式，计算

$$o=f^{31} \bmod g_r$$

式中，$o$ 可看成二元数组，即 $S$ 盒的 8 比特输出。

4 个 $S$ 盒的 32 比特输出经过 $P$ 置换后可变成层函数 $F$ 的输出，如表 1.6.2 所示。

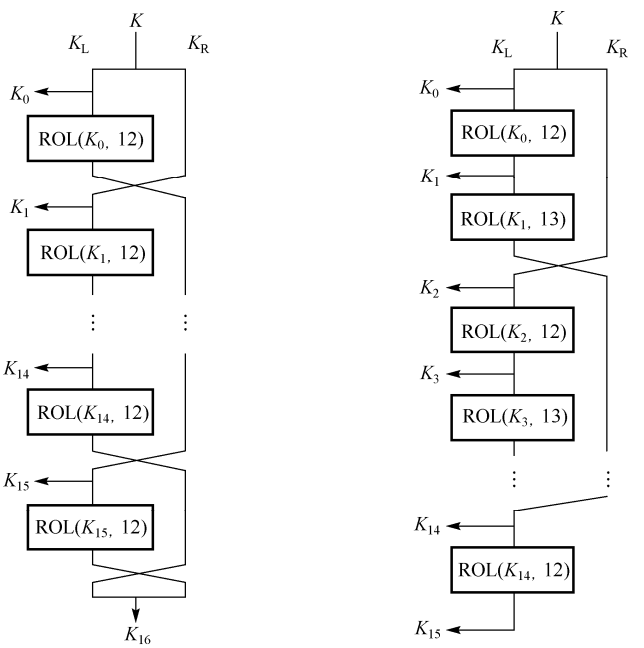
表 1.6.2　层函数 $F$ 的输出

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 31 | 23 | 15 | 7 | 30 | 22 | 14 | 6 |
| 29 | 21 | 13 | 5 | 28 | 20 | 12 | 4 |
| 27 | 19 | 11 | 3 | 26 | 18 | 10 | 2 |
| 25 | 17 | 9 | 1 | 24 | 16 | 8 | 0 |

LOKI89 加密算法的层密钥是通过将密钥分成左右两半，并且经过变换和循环移位生成的，记 $K=K_L|K_R$，ROL(·,$n$)表示循环左移 $n$ 位，相关伪代码如下：

```
for(i=0; i<16; i+=2)
{
    Ki=KL
    KL=ROL(KL,12)
    Ki+1=KR
    KR=ROL(KR,12)
}
    KL=ROL(K14,12)
    KR=ROL(K15,12)
    K16=KL|KR
```

LOKI89 加密算法和 LOKI91 加密算法的层密钥生成规律如图 1.6.1 所示。




(a) LOKI89加密算法　　(b) LOKI91加密算法

图 1.6.1　LOKI89 加密算法和 LOKI91 加密算法的层密钥生成规律

### 1.6.2　LOKI91 加密算法

LOKI91 加密算法的层密钥生成的相关伪代码如下：

```
for(i=0; i<16; i+=4)
{
    K_i=K_L
    K_L=ROL(K_L,12)
    K_{i+1}=K_L
    K_L=ROL(K_L,13)
    K_{i+2}=K_R
    K_R=ROL(K_R,12)
    K_{i+3}=K_R
    K_R=ROL(K_R,13)
}
```

LOKI91 加密算法的结构框图和层密钥生成过程如图 1.6.2 所示。

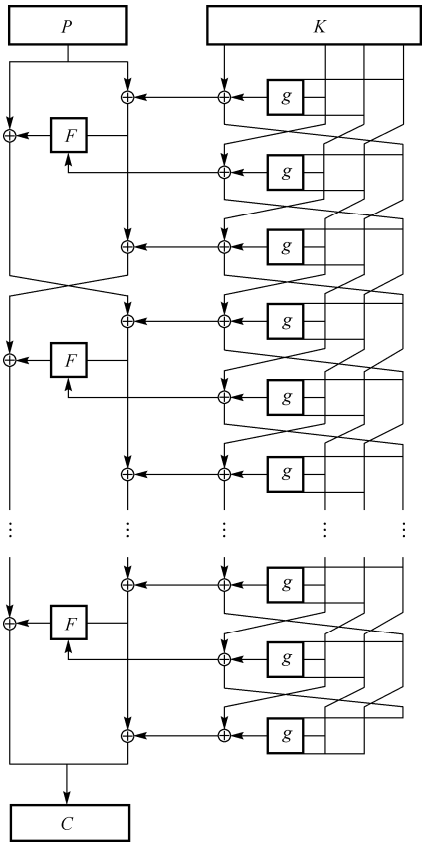



图 1.6.2　LOKI91 加密算法的结构框图和层密钥生成过程

将 128 比特的明文分组分成 64 比特的左右两部分，即 $L_0$ 和 $R_0$，然后进行如下伪代码所示的 16 层变换。

```
for(i=1; i<=12; i++)
{
    Ri=Li-1 ⊕ F(Ri-1+K3i-2,K3i-1);
    Li=Ri-1+K3i-2+K3i;
}
```

最后一层左右两部分不变换。

$K_{3i-2}$、$K_{3i-1}$、$K_{3i}$（$i$=1,2,…,16）为 48 个 64 比特的层密钥，层密钥生成规律如下：

（1）由输入密钥生成 4 个 64 比特的字 $K_{40} \| K_{30} \| K_{20} \| K_{10}$。如果输入密钥长度为 256 比特，则记为 4 个 64 比特的字 $K_a \| K_b \| K_c \| K_d$，则

$$K_{40} \| K_{30} \| K_{20} \| K_{10}=K_a \| K_b \| K_c \| K_d$$

如果输入密钥长度为 192 比特，则记为 3 个 64 比特的字 $K_a \| K_b \| K_c$，即

$$K_{40} \| K_{30} \| K_{20} \| K_{10}= K_a \| K_b \| K_c \| F(K_a,K_b)$$

如果输入密钥长度为 128 比特，则记为 2 个 64 比特的字 $K_a \| K_b$，即

$$K_{40} \| K_{30} \| K_{20} \| K_{10}= K_a \| K_b \| F(K_b,K_a) \| F(K_a,K_b)$$

（2）进行如下伪代码所示的 48 层变换。

```
for(i=1; i≤48; i++)
{
    Ki=K1,i=K4,i-1 ⊕ gi(K1,i-1,K3,i-1,K2,i-1);
    K4,i=K3,i-1;
    K3,i=K2,i-1;
    K2,i=K1,i-1;
}
```

其中

$$g_i(K_1,K_3,K_2)=F[K_1+K_3+(\delta \times i),K_2]$$

$$\delta = [\text{sqrt}(5)-1]\times 2^{63}=0\text{x}9\text{E}3779\text{B}97\text{F}4\text{A}7\text{C}15$$

函数 $F(A,B)$由五部分组成，如图 1.6.3 所示。

第一部分是一个由参数 $B$（即部分密钥）控制的比特置换，记为 KP($A$, $B$)。如果将 $A$ 分成 32 比特的左右两部分，即 $A_L$ 和 $A_R$，记 $B$ 的低位 32 比特为 $B_R$，则 KP($A$, $B$)就是下面的置换：当 $B$ 的第 $i$（$i$=1,2,…,32）比特为 1 时，$A_L$ 和 $A_R$ 的对应比特互换，用式子表示 KP($A$,$B$)：

$$\text{KP}(A,B)=\text{KP}(A_L \| A_R,B_R)=\{[(A_L\&\sim B_R)|(A_R \cdot B_R)] \| [(A_R\&\sim B_R)|(A_L \cdot B_R)]\}$$

式中，&～表示逐位进行与非运算，||表示连接，|表示逐位进行或运算，·表示逐位进行与运算。

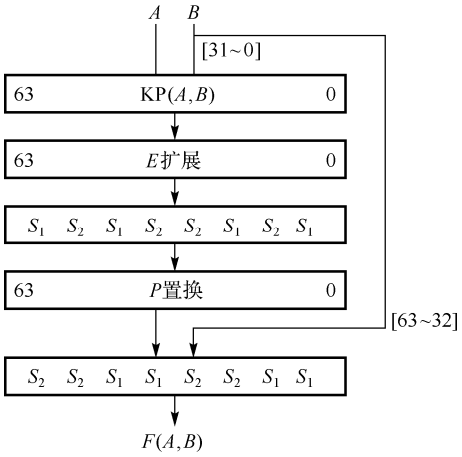



图 1.6.3　$F(A,B)$函数的组成示意图

第二部分是一个 $E$ 扩展，它将 64 比特扩展成 96 比特后作为 $S$ 盒的输入。$E$ 扩展规律如表 1.6.3 所示。

表 1.6.3　$E$ 扩展规律

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 1 | 0 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 |
| 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | | |
| 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 |
| 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | | |
| 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | | |
| 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | | |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

第三部分是 8 个 $S$ 盒，由 $S_1$ 和 $S_2$ 按 $S_1S_2S_1S_2S_2S_1S_2S_1$ 的顺序组成，$S_1$ 和 $S_2$ 的输入分别为 13 比特和 11 比特，输出都为 8 比特，由下列两个式子生成。

$$S_1(x)=[(x \oplus 0\text{x}1\text{fff})^3 \bmod 0\text{x}2911] \,\&\, 0\text{xff}$$

$$S_2(x)=[(x \oplus 0\text{x}7\text{ff})^3 \bmod 0\text{xAA}7] \,\&\, 0\text{xff}$$

上面两个式子分别是在有限域 $GF(2^{13})$和 $GF(2^{11})$中进行运算的。$E$ 扩展后的 96 比特从左到右依次作为 8 个 $S$ 盒的输入。

第四部分是一个固定的 $P$ 置换，如表 1.6.4 所示。

第五部分又是 8 个 $S$ 盒，由 $S_1$ 和 $S_2$ 按 $S_2S_2S_1S_1S_2S_2S_1S_1$ 的顺序组成。$P$ 置换的 64 比特输出从左到右依次作为 8 个 $S$ 盒的低 8 位输入，每个 $S$ 盒的高输入比特则由 $B$ 的高 32 比特（63～32 比特）从左到右按顺序组成。

表 1.6.4　*P* 置换

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 56 | 48 | 40 | 32 | 24 | 16 | 8 | 0 | 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 |
| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 | 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 60 | 52 | 44 | 36 | 28 | 20 | 12 | 4 | 61 | 53 | 45 | 37 | 29 | 21 | 13 | 5 |
| 62 | 54 | 46 | 38 | 30 | 22 | 14 | 6 | 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 |

## 1.7　GOST 加密算法

GOST 加密算法是俄罗斯现行的加密标准，在 1989 年 4 月 22 日由 1409 号决议通过，在 1990 年 7 月 1 日正式实施。

GOST 是 32 层 DES 型迭代分组密码，分组长度为 64 比特，密钥长度为 256 比特。

与 DES 加密算法相比，GOST 加密算法除了增加了迭代层数和密钥长度，还有下列主要差别。

（1）由初始密钥生成层密钥的过程十分简单。将 256 比特密钥分成 8 个 32 比特的字，记为 $K_0,K_1,\cdots,K_7$，则 32 层的层密钥分别为：

$$K_0\sim K_7,K_0\sim K_7,K_0\sim K_7,K_7\sim K_0$$

记为 $SK_1\sim SK_{32}$。

（2）层变换的差别是：①GOST 加密算法没有输入分组扩展；②GOST 加密算法与层密钥进行的是模 $2^{32}$ 加，而 DES 加密算法进行的是模 2 加；③GOST 加密算法用 8 个 $GF(2^4)\rightarrow GF(2^4)$的非线性代替；④GOST 加密算法用简单的循环移位代替不规则置换。

（3）*S* 盒保密，不同系统可用不同的 *S* 盒，从而使密钥为：

$$256+\log_2(16!)\times 8=610 \text{ 比特}$$

GOST 加密算法的 *F* 函数由三部分组成：

- 与 32 比特层密钥 $SK_i$ 进行模 $2^{32}$ 加；
- 经 8 个 4 比特 *S* 盒；
- 循环左移 11 比特。

输入的明文从低位到高位排列构成（$X_0,X_1$），$X_0$ 和 $X_1$ 各 32 比特，迭代计算为：

$$X_{i+1}=F(X_i)\oplus X_{i-1},\qquad i=1,\cdots,32$$

经过 32 层迭代可得（$X_{33}, X_{32}$），自低位到高位的输出就是密文。

GOST 加密算法有以下 4 种工作方式：

（1）简单代替方式，即电子密本（ECB）方式。

（2）乱数加密方式，即 64 比特的输出反馈方式 OFB(64)，以 64 比特同步信号作为初始向量。

（3）反馈乱数加密方式，即 64 比特的密文反馈方式 CFB(64)。

（4）验证方式，即按初始向量为 0 的密文分组连接方式进行计算，在最后一组输出中选 $L$ 比特（$L \leqslant 64$）作为消息验证码。

GOST 加密算法框图如图 1.7.1 所示。

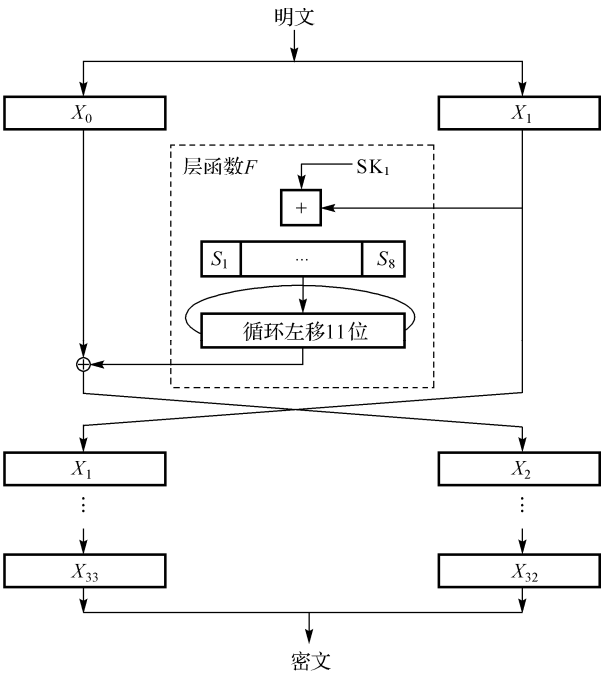



图 1.7.1　GOST 加密算法框图

## 1.8　BLOWFISH 加密算法

BLOWFISH 加密算法是由美国的 Bruce Schneier 于 1993 年提出的，属于分组密码，它采用平衡 Feistel 结构，分组长度、迭代层数和 DES 加密算法完全相同，结构框架和 DES 加密算法也很相似，只是其密钥长度可变，变化范围为 1～448 比特。

整个加密算法由密钥扩展和数据加密两个独立的部分组成。密钥扩展是由输入密钥生成 1042 个 32 比特的字的子密钥过程（下文所指的字均为 32 比特）。该过程工程量大，需要预先完成，将大批的子密钥存放在缓冲存储器中，以备数据加密时调用。在这 1042 个字的子密钥中，前 18 个字组成 $P$ 数组$\{P_1, P_2, \cdots, P_{18}\}$，后面的 1024 个字按顺序组成 4 个 $S$ 盒，每个 $S$ 盒由 256 个字组成，记为 $S_{ij}$（$i$=1,2,3,4；$j$=0,1,⋯,255）。

数据加密采用一个 16 层的平衡 Feistel 结构，每层由一次加乱、一个 $P$ 置换和一个 $S$ 盒代替组成，层内也只有简单的 xor 和模 $2^{32}$ 加运算。

输入的 64 比特明文平分成 32 比特的左右两部分，记为 $X_L$ 和 $X_R$。16 层加密过程可用下面的伪代码表示。

```
for i=1 to 16 do
{
    X_L=X_L ⊕ P_i
    X_R=X_R ⊕ F(X_L)
    交换 X_L 和 X_R
}
X_L=X_R ⊕ P_17
X_R=X_L ⊕ P_18
```

$X_L \| X_R$ 即密文。

层函数 $F$ 如下：将 $X_L$ 从左到右按字节分成 4 个值（$a$、$b$、$c$ 和 $d$），然后分别在 4 个 $S$ 盒（$S_1$、$S_2$、$S_3$、$S_4$）中进行代替，得到 4 个字 $S_{1a}$、$S_{2b}$、$S_{3c}$、$S_{4d}$，按下式对这 4 个字进行结合，可得到 $F$ 函数的输出。

$$F(X_L)=\{[(S_{1a}+S_{2b}) \bmod 2^{32}] \oplus S_{3c}\}+S_{4d} \bmod 2^{32}$$

从上面可以看出，BLOWFISH 加密算法的数据加密非常简洁，操作也非常简单，如果子密钥预先存储在高速缓冲存储器中，则 BLOWFISH 加密算法的实现要比 DES 加密算法快得多。

BLOWFISH 的脱密过程和加密过程完全一样，只需将 $P$ 数组逆序使用即可。

子密钥的生产也使用了 BLOWFISH 加密算法，具体生成方法如下：

（1）用一个随机序列（可以固定不变）初始化 $P$ 数组和 4 个 $S$ 盒。

（2）对密钥 $K$ 进行扩展，使扩展后的密钥不小于 18×32 比特，将扩展后的密钥与 $P$ 数组进行模 2 加得到新的数组。

（3）采用 BLOWFISH 加密算法对 64 比特的全 0 串进行加密，用最后输出的 $X_L$ 和 $X_R$ 替代 $P_1$ 和 $P_2$；然后将 $X_L$ 和 $X_R$ 作为明文，用更新后的子密钥和 BLOWFISH 加密算法进行加密，用最后输出的 $X_L$ 和 $X_R$ 替代 $P_3$ 和 $P_4$；重复此过程 521 次，更新 1042 个的子密钥。

和 DES 加密算法相比，BLOWFISH 加密算法减少了 $S$ 盒的数量，但是它的 $S$ 盒不是固定不变的，而是随密钥数据变化的，结构是未知的。4 个 $S$ 盒的输出经过模 2 加和模 $2^{32}$ 加结合后才作为层函数 $F$ 的输出，各层所加乱数各不相关，因此 BLOWFISH 加密算法能有效抵制线性攻击和差分分析攻击，具有极高的安全性。

## 1.9　Khufu 和 Khafre 快速软件加密算法

Khufu 和 Khafre 快速软件加密算法是由 R. C. Merkle 在 1990 年美洲密码年会上提出的，类似于 DES 的多层迭代分组密码，迭代层数规定为 8 的整数倍，输入和输出分组均为 64 比特，加密速度在 SUN 4/260 上可达 4～8 Mb/s。因为是软件加密算法，所以与硬件加密算法相比有更大的灵活性，其加密层数和密钥长度均没有具体规定，可根据被加密数据的价值，以及对安全性和加密速度的要求来确定。迭代层数可以是 8～64 之间的任意 8 的整数倍，一般用 16 层、24 层或 32 层。密钥长度一般为 64 比特的整数倍。

Khufu 和 Khafre 设计的基本思想是：面向软件、快速实现。当时 R.C. Merkle 认为还没有一种被大家普遍接受的软件加密算法，大部分商业加密软件都用软件仿效硬件实现 DES 加密算法。由于 DES 加密算法是面向硬件设计的，每层要并行进行 8 次查表（*S* 盒代替），然后进行一次置换。而对于软件来说，8 次查表也只能逐个进行，所以速度慢。R. C. Merkle 估计，用软件实现 DES 加密算法的速度为安全性相当的、面向软件加密算法的 1/10～1/5。

Khufu 加密算法和 Khafre 加密算法基本相同，略有差别。在加密时，前者的 *S* 盒由密钥决定，后者的 *S* 盒固定不变；在使用时，前者用于大量成批数据的加密，后者用于少量数据的加密。

我们先介绍 Khufu 加密算法。把数据（明文）分为左、右两半，各 32 比特，分别记为 *L* 和 *R*，加密的第一步是把明文与 64 比特的辅助密钥（Auxiliary Key，AK）进行模 2 加，然后进行 *n* 层变换，最后与另外 64 比特的辅助密钥进行模 2 加，最后得到输出密文。每层变换由一次查表和一次循环移位组成，把 *L* 最右边一个字节作为 *S* 盒的输入，输出 32 比特并与 *R* 进行模 2 加，然后 *L* 循环右移 1 个或多个字节，再变换左、右两部分，从而实现一层变换。Khufu 加密算法如图 1.9.1 所示。

*S* 盒是一个由 256 个 32 比特组成的 RAM 表，若把它看成 4 个长度为 256 字节的序列，则每列是 0～255 的一个排列。*S* 盒的输入是 1 字节，把它作为地址码，从 RAM 表中取出相应的 32 比特作为输出。每经过 8 层变换，*S* 盒就要更换一次，所以 *S* 盒的个数是由层数决定的。左部分 1～8 层的循环右移的字节数分别为 2、2、1、1、2、2、3、3，经过 8 层循环右移后，数据回到原来位置，然后进行 9～16 层、17～24 层…，不断重复上述循环右移的字节数。经过 8 层变化后，每个字节恰好作为一次 *S* 盒的输入。

R. C. Merkle 在 1990 年美洲密码年会上发表的文章中对 Khufu 加密算法的密钥量没做规定，只是谈到从初始密钥算出 128 比特的辅助密钥和每 8 层变化一次的 *S* 盒，并没有介绍怎样从初始密钥计算辅助密钥。各层的 *S* 盒是从一个初始 *S* 盒出发，在初始密钥参与的情况下，

按一定的算法生成的。初始 $S$ 盒也是一个 256×32 比特的 RAM 表，每个字节列都是 0～255 的一个排列，生成 $S$ 盒的过程可分为两步。

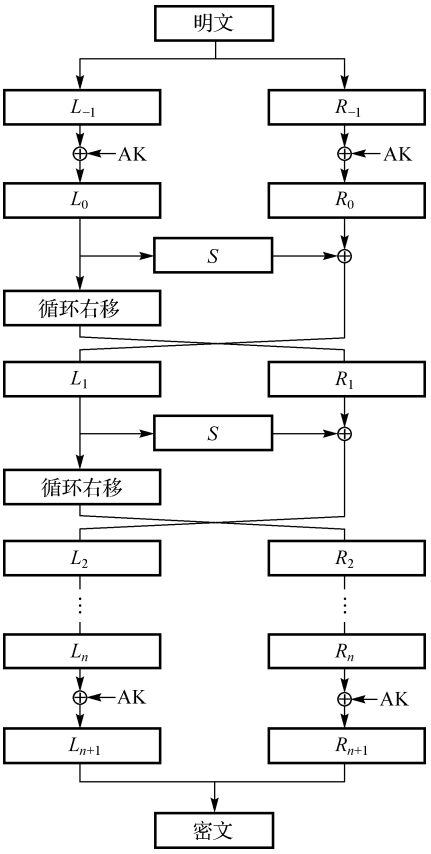



图 1.9.1　Khufu 加密算法

（1）生成一个足够长的伪随机字节序列。先从初始密钥得到 64 字节的初始值，采用 Khufu 加密算法，以分组连接的方式对这 64 字节进行加密，得到 64 个伪随机字节，再次加密后又可又得到 64 个伪随机字节…，直到伪随机字节序列足够长，在下面第（2）步中用它来决定第 $R[i]$个字节的值。

（2）由伪随机字节序列来产生初始 $S$ 盒。从初始 $S$ 盒的每个字节列的第 0 个字节开始到第 255 个字节，对每个字节进行一次对换，第 $i$ 个字节与该列的第 $R[i]$个字节对换，其中 $R[i]$是由第（1）步生成的相应伪随机字节决定的。经过这样变化后的 $S$ 盒仍然保持每列是 0～255 的一个排列，用上述方法可以生成足够多的 $S$ 盒。

Khufu 加密算法的每层变化使用了一个较大的非线性 $S$ 盒，$S$ 盒的输出与右半部分 $R$ 进

行模 2 加，经多层非线性变化后，达到了混乱的目的。又因为每层进行一次字节移位，经 8 层变化后，明文输入的每个字节都恰好用来查一次表，使变化后的每个密文的比特依赖于整个明文输入和密钥，达到了扩散的目的。

DES 加密算法每层要查 8 个 *S* 盒，进行一次 64 比特的置换。Khufu 加密算法针对软件实现的特点，把 *S* 盒的规模扩大，每层只查一次，并用循环移位来代替置换，这样同样达到了混乱和扩散的目的，但速度比 DES 加密算法快得多。

Khafre 加密算法与 Khufu 加密算法类似，不同的是 Khafre 加密算法用一组固定的 *S* 盒，称其为标准 *S* 盒。在加密少量数据或只加密一个 64 比特分组时，由于不必计算 *S* 盒，所以速度快。另外，因为 Khafre 加密算法的 *S* 盒与密钥无关，所以除了在算法的开头和结束数据要和 64 比特的密钥进行模 2 加，每经过 8 层变换，都要用 64 比特的密钥与数据进行一次模 2 加。此外，由于 *S* 盒是固定的，因此 Khafre 加密算法要用更多层的变化才能得到与 Khufu 加密算法相同的安全性。

## 1.10　CAST 加密算法

CAST 是由加拿大 Entrust Technologies 公司的 Carlisle Adams 和 Stafford Tavares 提出的一种 Feistel 型分组密码，它包括 CAST-128 加密算法和 CAST-256 加密算法，128 和 256 分别指两个算法的最大密钥长度。

CAST-128 加密算法是于 1997 年提出的，其分组长度、迭代层数、加密过程都非常类似于 DES，同样是分成左右两半，右边经层函数运算后与左边进行模 2 加，再交换；不同之处在于，CAST 加密算法的层函数有三种，轮流使用，每层使用一种，层函数包括加（模 2 加或减、模 $2^{32}$ 加或减），有密钥控制的循环移位和 *S* 盒代替（$2^8 \rightarrow 2^{32}$ 查表）。CAST-128 加密算法的输入和输出分组长度为 64 比特，输入密钥长度 *L* 可变，$L=40+i\times 8$ 比特（$i=0,1,2,\cdots,11$），迭代层数 *n* 有两种可能值，当 $L\leqslant 80$ 时，$n=12$；当 $L>80$ 时，$n=16$。当 $L<128$ 时，在密钥的低位补“0”，补足 128 比特。

CAST-256 加密算法是作为 AES 的候选加密算法提出来的，是 CAST-128 加密算法的一种改进形式，其分组长度为 128 比特，密钥可以为 128、160、192、224、256 比特，迭代层数为 12。

下面分别介绍这两种算法。

### 1.10.1　CAST-128 加密算法

CAST-128 加密算法可分成两部分来描述：由输入密钥生成层密钥，以及加密过程。

### 1．由输入密钥生成层密钥

（1）$n$ 层迭代变换需要 $n$ 对层密钥，记为 $\{K_m^{(i)},K_r^{(i)}\}$，$i$=1,2,⋯,$n$，其中 $K_m$ 为 32 比特，称为 Masking Key；$K_r$ 为 5 比特，称为 Rotation Key。由输入密钥生成层密钥的方法如下：

将输入密钥的 16 个字节由高到低依次记为 $X_0,X_1,\cdots,X_E,X_F$，将中间结果的 16 字节由高到低依次记为 $Z_0,Z_1,\cdots,Z_E,Z_F$，$S_i$[ ]代表第 $i$ 个 $S$ 盒。

$$Z_0Z_1Z_2Z_3=X_0X_1X_2X_3\oplus S_5[X_D]\oplus S_6[X_F]\oplus S_7[X_C]\oplus S_8[X_E]\oplus S_7[X_8]$$
$$Z_4Z_5Z_6Z_7=X_8X_9X_AX_B\oplus S_5[Z_0]\oplus S_6[Z_2]\oplus S_7[Z_1]\oplus S_8[Z_3]\oplus S_8[X_A]$$
$$Z_8Z_9Z_AZ_B=X_CX_DX_EX_F\oplus S_5[Z_7]\oplus S_6[Z_6]\oplus S_7[Z_5]\oplus S_8[Z_4]\oplus S_5[X_9]$$
$$Z_CZ_DZ_EZ_F=X_4X_5X_6X_7\oplus S_5[Z_A]\oplus S_6[Z_9]\oplus S_7[Z_B]\oplus S_8[Z_8]\oplus S_6[X_B]$$
$$K_1=S_5[Z_8]\oplus S_6[Z_9]\oplus S_7[Z_7]\oplus S_8[Z_6]\oplus S_5[Z_2]$$
$$K_2=S_5[Z_A]\oplus S_6[Z_B]\oplus S_7[Z_5]\oplus S_8[Z_4]\oplus S_6[Z_6]$$
$$K_3=S_5[Z_C]\oplus S_6[Z_D]\oplus S_7[Z_3]\oplus S_8[Z_2]\oplus S_7[Z_9]$$
$$K_4=S_5[Z_E]\oplus S_6[Z_F]\oplus S_7[Z_1]\oplus S_8[Z_0]\oplus S_8[Z_C]$$
$$X_0X_1X_2X_3=Z_8Z_9Z_AZ_B\oplus S_5[Z_5]\oplus S_6[Z_7]\oplus S_7[Z_4]\oplus S_8[Z_6]\oplus S_7[Z_0]$$
$$X_4X_5X_6X_7=Z_0Z_1Z_2Z_3\oplus S_5[X_0]\oplus S_6[X_2]\oplus S_7[X_1]\oplus S_8[X_3]\oplus S_8[Z_2]$$
$$X_8X_9X_AX_B=Z_4Z_5Z_6Z_7\oplus S_5[X_7]\oplus S_6[X_6]\oplus S_7[X_5]\oplus S_8[X_4]\oplus S_5[Z_1]$$
$$X_CX_DX_EX_F=Z_CZ_DZ_EZ_F\oplus S_5[X_A]\oplus S_6[X_9]\oplus S_7[X_B]\oplus S_8[X_8]\oplus S_6[Z_3]$$
$$K_5=S_5[X_3]\oplus S_6[X_2]\oplus S_7[X_C]\oplus S_8[X_D]\oplus S_5[X_8]$$
$$K_6=S_5[X_1]\oplus S_6[X_0]\oplus S_7[X_E]\oplus S_8[X_F]\oplus S_6[X_D]$$
$$K_7=S_5[X_7]\oplus S_6[X_6]\oplus S_7[X_8]\oplus S_8[X_9]\oplus S_7[X_3]$$
$$K_8=S_5[X_5]\oplus S_6[X_4]\oplus S_7[X_A]\oplus S_8[X_B]\oplus S_8[X_7]$$
$$Z_0Z_1Z_2Z_3=X_0X_1X_2X_3\oplus S_5[X_D]\oplus S_6[X_F]\oplus S_7[X_C]\oplus S_8[X_E]\oplus S_7[X_8]$$
$$Z_4Z_5Z_6Z_7=X_8X_9X_AX_B\oplus S_5[Z_0]\oplus S_6[Z_2]\oplus S_7[Z_1]\oplus S_8[Z_3]\oplus S_8[X_A]$$
$$Z_8Z_9Z_AZ_B=X_CX_DX_EX_F\oplus S_5[Z_7]\oplus S_6[Z_6]\oplus S_7[Z_5]\oplus S_8[Z_4]\oplus S_5[X_9]$$
$$Z_CZ_DZ_EZ_F=X_4X_5X_6X_7\oplus S_5[Z_A]\oplus S_6[Z_9]\oplus S_7[Z_B]\oplus S_8[Z_8]\oplus S_6[X_B]$$
$$K_9=S_5[Z_3]\oplus S_6[Z_2]\oplus S_7[Z_C]\oplus S_8[Z_D]\oplus S_5[Z_9]$$
$$K_{10}=S_5[Z_1]\oplus S_6[Z_0]\oplus S_7[Z_E]\oplus S_8[Z_F]\oplus S_6[Z_C]$$
$$K_{11}=S_5[Z_7]\oplus S_6[Z_6]\oplus S_7[Z_8]\oplus S_8[Z_9]\oplus S_7[Z_2]$$
$$K_{12}=S_5[Z_5]\oplus S_6[Z_4]\oplus S_7[Z_A]\oplus S_8[Z_B]\oplus S_8[Z_6]$$
$$X_0X_1X_2X_3=Z_8Z_9Z_AZ_B\oplus S_5[Z_5]\oplus S_6[Z_7]\oplus S_7[Z_4]\oplus S_8[Z_6]\oplus S_7[Z_0]$$
$$X_4X_5X_6X_7=Z_0Z_1Z_2Z_3\oplus S_5[X_0]\oplus S_6[X_2]\oplus S_7[X_1]\oplus S_8[X_3]\oplus S_8[Z_2]$$
$$X_8X_9X_AX_B=Z_4Z_5Z_6Z_7\oplus S_5[X_7]\oplus S_6[X_6]\oplus S_7[X_5]\oplus S_8[X_4]\oplus S_5[Z_1]$$

$$X_C X_D X_E X_F = Z_C Z_D Z_E Z_F \oplus S_5[X_A] \oplus S_6[X_9] \oplus S_7[X_B] \oplus S_8[X_8] \oplus S_6[Z_3]$$

$$K_{13} = S_5[X_8] \oplus S_6[X_9] \oplus S_7[X_7] \oplus S_8[X_6] \oplus S_5[X_3]$$

$$K_{14} = S_5[X_A] \oplus S_6[X_B] \oplus S_7[X_5] \oplus S_8[X_4] \oplus S_6[X_7]$$

$$K_{15} = S_5[X_C] \oplus S_6[X_D] \oplus S_7[X_3] \oplus S_8[X_2] \oplus S_7[X_8]$$

$$K_{16} = S_5[X_E] \oplus S_6[X_F] \oplus S_7[X_1] \oplus S_8[X_0] \oplus S_8[X_D]$$

$K_1$～$K_{16}$ 分别用于 16 层的 $K_m$，然后用当前的 $X_0, X_1, \cdots, X_E, X_F$ 值，采用与上面一样的方法生成 16 层的 $K_r$，$K_r^{(i)} = K_i \,\&\, 31$。

### 2．加密过程

（1）将 64 比特的明文平分成左右两半，记为 $L_0$ 和 $R_0$。

（2）采用下述伪代码进行处理。

```
for(i=1;i<=16; i++)
{
    Li=Ri-1;
    Ri=Li-1 ⊕ F(Ri-1,Km(i),Kr(i));
}
```

（3）交换 $L_{16}$ 与 $R_{16}$，即可输出的密文。

CAST-128 加密算法的一个突出特点是每层的层函数 $F$ 不一样，层函数 $F$ 有以下三种类型。

第一种：

$$I = (K_m^{(i)} + D) <<< K_r^{(i)}$$

$$F_1 = [(S_1[I_a] \oplus S_2[I_b]) - S_3[I_c]] + S_4[I_d]$$

第二种：

$$I = (K_m^{(i)} \oplus D) <<< K_r^{(i)}$$

$$F_2 = [(S_1[I_a] - S_2[I_b]) + S_3[I_c]] \oplus S_4[I_d]$$

第三种：

$$I = (K_m^{(i)} - D) <<< K_r^{(i)}$$

$$F_3 = [(S_1[I_a] + S_2[I_b]) \oplus S_3[I_c]] - S_4[I_d]$$

式中，$D$ 表示层函数 $F$ 的输入数据，$I_a$、$I_b$、$I_c$、$I_d$ 分别表示 $I$ 的 4 个字节，“+”“–”运算均为模 $2^{32}$ 运算，“<<<”表示循环左移位。

从第 1 层到第 16 层依次重复使用层函数 $F$ 的三种类型，即第 1、4、7、10、13、16 层用层函数 $F$ 的第一种类型，第 2、5、8、11、14 层用层函数 $F$ 的第二种类型，第 3、6、9、12、15 层用层函数 $F$ 的第三种类型。

CAST-128 加密算法使用了 8 个 $S$ 盒，每个 $S$ 盒的输入都为 8 比特，输出为 32 比特，$S_1$、$S_2$、$S_3$ 和 $S_4$ 仅用于层函数 $F$，$S_5$、$S_6$、$S_7$ 和 $S_8$ 仅用于生成层密钥。

CAST-128 加密算法采用标准的 DES 型框架，每层只变换层输入的一半。如果把变换完整个层输入称为一层的话，那么 CAST-128 加密算法只有 8 层。

### 1.10.2 CAST-256 加密算法

CAST-256 加密算法的分组长度为 128 比特，记为 $A$、$B$、$C$、$D$ 四个 32 比特的字，每次只变换其中的一个字。如果把变换完整个层输入称为一层的话，那么 CAST-256 加密算法有 12 层。三种层函数 $F$ 和 8 个 $S$ 盒都和 CAST-128 加密算法一样，将三个层函数记为 $F_1$、$F_2$、$F_3$。CAST-256 加密算法的层密钥也与 CAST-128 加密算法一样，只是层密钥的生成方法不同，CAST-256 加密算法采用自身的算法来生成层密钥，并没有利用后面的 4 个 $S$ 盒。

将第 $i$ 层的层密钥记为：

$$K_{\mathrm{r}}^{(i)} = \{K_{\mathrm{r0}}^{(i)}, K_{\mathrm{r1}}^{(i)}, K_{\mathrm{r2}}^{(i)}, K_{\mathrm{r3}}^{(i)}\}$$
$$K_{\mathrm{m}}^{(i)} = \{K_{\mathrm{m0}}^{(i)}, K_{\mathrm{m1}}^{(i)}, K_{\mathrm{m2}}^{(i)}, K_{\mathrm{m3}}^{(i)}\}$$

令 $\beta$ 表示 128 比特的输入，CAST-256 加密算法的伪代码如下：

```
for(i=0; i<6; i++)
    β←Q(β);
for(i=6; i<12; i++)
    β←Q̄ (β);
```

经过上述伪代码处理之后，得到的 $\beta$ 就是密文。

前 6 层使用的层变换 $\beta \leftarrow Q(\beta)$为：

$$C = C \oplus F_1(D, K_{\mathrm{r0}}^{(i)}, K_{\mathrm{m0}}^{(i)})$$
$$B = B \oplus F_2(C, K_{\mathrm{r1}}^{(i)}, K_{\mathrm{m1}}^{(i)})$$
$$A = A \oplus F_3(B, K_{\mathrm{r2}}^{(i)}, K_{\mathrm{m2}}^{(i)})$$
$$D = D \oplus F_1(A, K_{\mathrm{r3}}^{(i)}, K_{\mathrm{m3}}^{(i)})$$

后 6 层使用的层变换 $\beta \leftarrow \overline{Q}(\beta)$为：

$$D = D \oplus F_1(A, K_{\mathrm{r3}}^{(i)}, K_{\mathrm{m3}}^{(i)})$$
$$A = A \oplus F_3(B, K_{\mathrm{r2}}^{(i)}, K_{\mathrm{m2}}^{(i)})$$
$$B = B \oplus F_2(C, K_{\mathrm{r1}}^{(i)}, K_{\mathrm{m1}}^{(i)})$$
$$C = C \oplus F_1(D, K_{\mathrm{r0}}^{(i)}, K_{\mathrm{m0}}^{(i)})$$

CAST-256 加密算法的两种层变换如图 1.10.1 所示。

从图 1.10.1 中可以清楚地看出，后 6 层所用的层变换和前 6 层所用的层变换是互逆的，这样可以采用同一套装置来加/脱密。

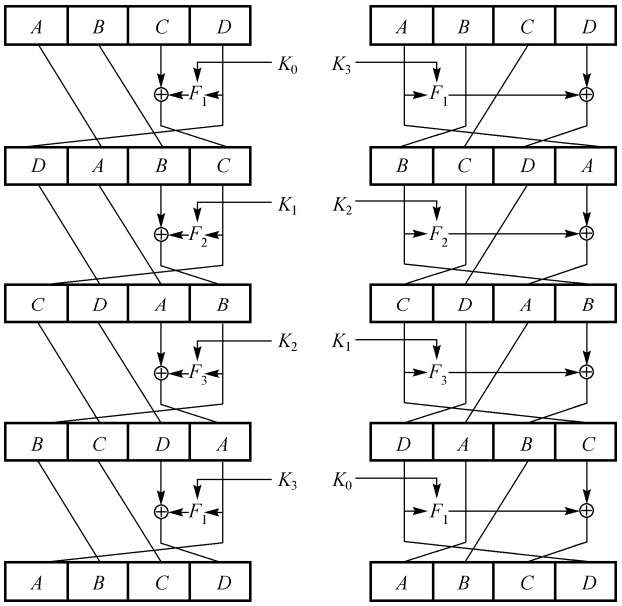



图 1.10.1　CAST-256 加密算法的两种层变换

CAST-256 加密算法有五种密钥长度，但实际的层密钥生成过程是按 256 比特的密钥长度设计的，如果密钥长度不足 256 比特，则在后面补足 0。将 256 比特的密钥记为 $K=(ABCDEFGH)$，其中 $A$、$B$、$C$、$D$、$E$、$F$、$G$、$H$ 为 8 个 32 比特的字，层密钥生成方法的伪代码如下：

```
for(i=0; i<12; i++)
{
    K←ω2i(K);
    K←ω2i+1(K);
    K_r^(i) ={ K_r0^(i), K_r1^(i), K_r2^(i), K_r3^(i) }
        ={A&31,C&31,E&31,G&31);
    K_m^(i) ={ K_m0^(i), K_m1^(i), K_m2^(i), K_m3^(i) }
        ={H,F,D,B);
}
```

其中，变换 $K\leftarrow\omega_i(K)$的定义如下：

$$G = G \oplus F_1(H, T_{r0}^{(i)}, T_{m0}^{(i)})$$
$$F = F \oplus F_2(G, T_{r1}^{(i)}, T_{m1}^{(i)})$$
$$E = E \oplus F_3(F, T_{r2}^{(i)}, T_{m2}^{(i)})$$

$$D = D \oplus F_1(E, T_{r3}^{(i)}, T_{m3}^{(i)})$$
$$C = C \oplus F_2(D, T_{r4}^{(i)}, T_{m4}^{(i)})$$
$$B = B \oplus F_3(C, T_{r5}^{(i)}, T_{m5}^{(i)})$$
$$A = A \oplus F_1(B, T_{r6}^{(i)}, T_{m6}^{(i)})$$
$$H = H \oplus F_2(A, T_{r7}^{(i)}, T_{m7}^{(i)})$$

式中，$T_{ri}^{(i)}$ 和 $T_{mi}^{(i)}$ 为 Rotation Key 和 Masking Key，它们的值生成过程伪代码如下：

```
    cm=2^30 √2 =0x5a827999;
    mm=2^30 √3 =0x6ed9eba1;
    cr=19;
    mr=17;
for(i=0; i<24; i++)
    for(j=0; j<8; j++)
    {
        T_mj^(i) =cm;
        cm=(cm+mm) mod 2^32;
        T_rj^(i) =cr;
        cr=(cr+mr) mod 32;
    }
```

CAST 加密算法中的 $S$ 盒如下：

$S_1$：

```
30fb40d4 9fa0ff0b 6beccd2f 3f258c7a 1e213f2f 9c004dd3 6003e540 cf9fc949
bfd4af27 88bbbdb5 e2034090 98d09675 6e63a0e0 15c361d2 c2e7661d 22d4ff8e
28683b6f c07fd059 ff2379c8 775f50e2 43c340d3 df2f8656 887ca41a a2d2bd2d
a1c9e0d6 346c4819 61b76d87 22540f2f 2abe32e1 aa54166b 22568e3a a2d341d0
66db40c8 a784392f 004dff2f 2db9d2de 97943fac 4a97c1d8 527644b7 b5f437a7
b82cbaef d751d159 6ff7f0ed 5a097a1f 827b68d0 90ecf52e 22b0c054 bc8e5935
4b6d2f7f 50bb64a2 d2664910 bee5812d b7332290 e93b159f b48ee411 4bff345d
fd45c240 ad31973f c4f6d02e 55fc8165 d5b1caad a1ac2dae a2d4b76d c19b0c50
882240f2 0c6e4f38 a4e4bfd7 4f5ba272 564c1d2f c59c5319 b949e354 b04669fe
b1b6ab8a c71358dd 6385c545 110f935d 57538ad5 6a390493 e63d37e0 2a54f6b3
3a787d5f 6276a0b5 19a6fcdf 7a42206a 29f9d4d5 f61b1891 bb72275e aa508167
38901091 c6b505eb 84c7cb8c 2ad75a0f 874a1427 a2d1936b 2ad286af aa56d291
d7894360 425c750d 93b39e26 187184c9 6c00b32d 73e2bb14 a0bebc3c 54623779
64459eab 3f328b82 7718cf82 59a2cea6 04ee002e 89fe78e6 3fab0950 325ff6c2
81383f05 6963c5c8 76cb5ad6 d49974c9 ca180dcf 380782d5 c7fa5cf6 8ac31511
35e79e13 47da91d0 f40f9086 a7e2419e 31366241 051ef495 aa573b04 4a805d8d
548300d0 00322a3c bf64cddf ba57a68e 75c6372b 50afd341 a7c13275 915a0bf5
6b54bfab 2b0b1426 ab4cc9d7 449ccd82 f7fbf265 ab85c5f3 1b55db94 aad4e324
```

```
cfa4bd3f 2deaa3e2 9e204d02 c8bd25ac eadf55b3 d5bd9e98 e31231b2 2ad5ad6c
954329de adbe4528 d8710f69 aa51c90f aa786bf6 22513f1e aa51a79b 2ad344cc
7b5a41f0 d37cfbad 1b069505 41ece491 b4c332e6 032268d4 c9600acc ce387e6d
bf6bb16c 6a70fb78 0d03d9c9 d4df39de e01063da 4736f464 5ad328d8 b347cc96
75bb0fc3 98511bfb 4ffbcc35 b58bcf6a e11f0abc bfc5fe4a a70aec10 ac39570a
3f04442f 6188b153 e0397a2e 5727cb79 9ceb418f 1cacd68d 2ad37c96 0175cb9d
c69dff09 c75b65f0 d9db40d8 ec0e7779 4744ead4 b11c3274 dd24cb9e 7e1c54bd
f01144f9 d2240eb1 9675b3fd a3ac3755 d47c27af 51c85f4d 56907596 a5bb15e6
580304f0 ca042cf1 011a37ea 8dbfaadb 35ba3e4a 3526ffa0 c37b4d09 bc306ed9
98a52666 5648f725 ff5e569d 0ced63d0 7c63b2cf 700b45e1 d5ea50f1 85a92872
af1fbda7 d4234870 a7870bf3 2d3b4d79 42e04198 0cd0ede7 26470db8 f881814c
474d6ad7 7c0c5e5c d1231959 381b7298 f5d2f4db ab838653 6e2f1e23 83719c9e
bd91e046 9a56456e dc39200c 20c8c571 962bda1c e1e696ff b141ab08 7cca89b9
1a69e783 02cc4843 a2f7c579 429ef47d 427b169c 5ac9f049 dd8f0f00 5c8165bf
```

$S_2$：

```
1f201094 ef0ba75b 69e3cf7e 393f4380 fe61cf7a eec5207a 55889c94 72fc0651
ada7ef79 4e1d7235 d55a63ce de0436ba 99c430ef 5f0c0794 18dcdb7d a1d6eff3
a0b52f7b 59e83605 ee15b094 e9ffd909 dc440086 ef944459 ba83ccb3 e0c3cdfb
d1da4181 3b092ab1 f997f1c1 a5e6cf7b 01420ddb e4e7ef5b 25a1ff41 e180f806
1fc41080 179bee7a d37ac6a9 fe5830a4 98de8b7f 77e83f4e 79929269 24fa9f7b
e113c85b acc40083 d7503525 f7ea615f 62143154 0d554b63 5d681121 c866c359
3d63cf73 cee234c0 d4d87e87 5c672b21 071f6181 39f7627f 361e3084 e4eb573b
602f64a4 d63acd9c 1bbc4635 9e81032d 2701f50c 99847ab4 a0e3df79 ba6cf38c
10843094 2537a95e f46f6ffe a1ff3b1f 208cfb6a 8f458c74 d9e0a227 4ec73a34
fc884f69 3e4de8df ef0e0088 3559648d 8a45388c 1d804366 721d9bfd a58684bb
e8256333 844e8212 128d8098 fed33fb4 ce280ae1 27e19ba5 d5a6c252 e49754bd
c5d655dd eb667064 77840b4d a1b6a801 84db26a9 e0b56714 21f043b7 e5d05860
54f03084 066ff472 a31aa153 dadc4755 b5625dbf 68561be6 83ca6b94 2d6ed23b
eccf01db a6d3d0ba b6803d5c af77a709 33b4a34c 397bc8d6 5ee22b95 5f0e5304
81ed6f61 20e74364 b45e1378 de18639b 881ca122 b96726d1 8049a7e8 22b7da7b
5e552d25 5272d237 79d2951c c60d894c 488cb402 1ba4fe5b a4b09f6b 1ca815cf
a20c3005 8871df63 b9de2fcb 0cc6c9e9 0beeff53 e3214517 b4542835 9f63293c
ee41e729 6e1d2d7c 50045286 1e6685f3 f33401c6 30a22c95 31a70850 60930f13
73f98417 a1269859 ec645c44 52c877a9 cdff33a6 a02b1741 7cbad9a2 2180036f
50d99c08 cb3f4861 c26bd765 64a3f6ab 80342676 25a75e7b e4e6d1fc 20c710e6
cdf0b680 17844d3b 31eef84d 7e0824e4 2ccb49eb 846a3bae 8ff77888 ee5d60f6
7af75673 2fdd5cdb a11631c1 30f66f43 b3faec54 157fd7fa ef8579cc d152de58
db2ffd5e 8f32ce19 306af97a 02f03ef8 99319ad5 c242fa0f a7e3ebb0 c68e4906
b8da230c 80823028 dcdef3c8 d35fb171 088a1bc8 bec0c560 61a3c9e8 bca8f54d
c72feffa 22822e99 82c570b4 d8d94e89 8b1c34bc 301e16e6 273be979 b0ffeaa6
61d9b8c6 00b24869 b7ffce3f 08dc283b 43daf65a f7e19798 7619b72f 8f1c9ba4
```

```
dc8637a0 16a7d3b1 9fc393b7 a7136eeb c6bcc63e 1a513742 ef6828bc 520365d6
2d6a77ab 3527ed4b 821fd216 095c6e2e db92f2fb 5eea29cb 145892f5 91584f7f
5483697b 2667a8cc 85196048 8c4bacea 833860d4 0d23e0f9 6c387e8a 0ae6d249
b284600c d835731d dcb1c647 ac4c56ea 3ebd81b3 230eabb0 6438bc87 f0b5b1fa
8f5ea2b3 fc184642 0a036b7a 4fb089bd 649da589 a345415e 5c038323 3e5d3bb9
43d79572 7e6dd07c 06dfdf1e 6c6cc4ef 7160a539 73bfbe70 83877605 4523ecf1
```

$S_3$：

```
8defc240 25fa5d9f eb903dbf e810c907 47607fff 369fe44b 8c1fc644 aececa90
beb1f9bf eefbcaea e8cf1950 51df07ae 920e8806 f0ad0548 e13c8d83 927010d5
11107d9f 07647db9 b2e3e4d4 3d4f285e b9afa820 fade82e0 a067268b 8272792e
553fb2c0 489ae22b d4ef9794 125e3fbc 21fffcee 825b1bfd 9255c5ed 1257a240
4e1a8302 bae07fff 528246e7 8e57140e 3373f7bf 8c9f8188 a6fc4ee8 c982b5a5
a8c01db7 579fc264 67094f31 f2bd3f5f 40fff7c1 1fb78dfc 8e6bd2c1 437be59b
99b03dbf b5dbc64b 638dc0e6 55819d99 a197c81c 4a012d6e c5884a28 ccc36f71
b843c213 6c0743f1 8309893c 0feddd5f 2f7fe850 d7c07f7e 02507fbf 5afb9a04
a747d2d0 1651192e af70bf3e 58c31380 5f98302e 727cc3c4 0a0fb402 0f7fef82
8c96fdad 5d2c2aae 8ee99a49 50da88b8 8427f4a0 1eac5790 796fb449 8252dc15
efbd7d9b a672597d ada840d8 45f54504 fa5d7403 e83ec305 4f91751a 925669c2
23efe941 a903f12e 60270df2 0276e4b6 94fd6574 927985b2 8276dbcb 02778176
f8af918d 4e48f79e 8f616ddf e29d840e 842f7d83 340ce5c8 96bbb682 93b4b148
ef303cab 984faf28 779faf9b 92dc560d 224d1e20 8437aa88 7d29dc96 2756d3dc
8b907cee b51fd240 e7c07ce3 e566b4a1 c3e9615e 3cf8209d 6094d1e3 cd9ca341
5c76460e 00ea983b d4d67881 fd47572c f76cedd9 bda8229c 127dadaa 438a074e
1f97c090 081bdb8a 93a07ebe b938ca15 97b03cff 3dc2c0f8 8d1ab2ec 64380e51
68cc7bfb d90f2788 12490181 5de5ffd4 dd7ef86a 76a2e214 b9a40368 925d958f
4b39fffa ba39aee9 a4ffd30b faf7933b 6d498623 193cbcfa 27627545 825cf47a
61bd8ba0 d11e42d1 cead04f4 127ea392 10428db7 8272a972 9270c4a8 127de50b
285ba1c8 3c62f44f 35c0eaa5 e805d231 428929fb b4fcdf82 4fb66a53 0e7dc15b
1f081fab 108618ae fcfd086d f9ff2889 694bcc11 236a5cae 12deca4d 2c3f8cc5
d2d02dfe f8ef5896 e4cf52da 95155b67 494a488c b9b6a80c 5c8f82bc 89d36b45
3a609437 ec00c9a9 44715253 0a874b49 d773bc40 7c34671c 02717ef6 4feb5536
a2d02fff d2bf60c4 d43f03c0 50b4ef6d 07478cd1 006e1888 a2e53f55 b9e6d4bc
a2048016 97573833 d7207d67 de0f8f3d 72f87b33 abcc4f33 7688c55d 7b00a6b0
947b0001 570075d2 f9bb88f8 8942019e 4264a5ff 856302e0 72dbd92b ee971b69
6ea22fde 5f08ae2b af7a616d e5c98767 cf1febd2 61efc8c2 f1ac2571 cc8239c2
67214cb8 b1e583d1 b7dc3e62 7f10bdce f90a5c38 0ff0443d 606e6dc6 60543a49
5727c148 2be98a1d 8ab41738 20e1be24 af96da0f 68458425 99833be5 600d457d
282f9350 8334b362 d91d1120 2b6d8da0 642b1e31 9c305a00 52bce688 1b03588a
f7baefd5 4142ed9c a4315c11 83323ec5 dfef4636 a133c501 e9d3531c ee353783
```

$S_4$：

```
9db30420 1fb6e9de a7be7bef d273a298 4a4f7bdb 64ad8c57 85510443 fa020ed1
```

```
7e287aff e60fb663 095f35a1 79ebf120 fd059d43 6497b7b1 f3641f63 241e4adf
28147f5f 4fa2b8cd c9430040 0cc32220 fdd30b30 c0a5374f 1d2d00d9 24147b15
ee4d111a 0fca5167 71ff904c 2d195ffe 1a05645f 0c13fefe 081b08ca 05170121
80530100 e83e5efe ac9af4f8 7fe72701 d2b8ee5f 06df4261 bb9e9b8a 7293ea25
ce84ffdf f5718801 3dd64b04 a26f263b 7ed48400 547eebe6 446d4ca0 6cf3d6f5
2649abdf aea0c7f5 36338cc1 503f7e93 d3772061 11b638e1 72500e03 f80eb2bb
abe0502e ec8d77de 57971e81 e14f6746 c9335400 6920318f 081dbb99 ffc304a5
4d351805 7f3d5ce3 a6c866c6 5d5bcca9 daec6fea 9f926f91 9f46222f 3991467d
a5bf6d8e 1143c44f 43958302 d0214eeb 022083b8 3fb6180c 18f8931e 281658e6
26486e3e 8bd78a70 7477e4c1 b506e07c f32d0a25 79098b02 e4eabb81 28123b23
69dead38 1574ca16 df871b62 211c40b7 a51a9ef9 0014377b 041e8ac8 09114003
bd59e4d2 e3d156d5 4fe876d5 2f91a340 557be8de 00eae4a7 0ce5c2ec 4db4bba6
e756bdff dd3369ac ec17b035 06572327 99afc8b0 56c8c391 6b65811c 5e146119
6e85cb75 be07c002 c2325577 893ff4ec 5bbfc92d d0ec3b25 b7801ab7 8d6d3b24
20c763ef c366a5fc 9c382880 0ace3205 aac9548a eca1d7c7 041afa32 1d16625a
6701902c 9b757a54 31d477f7 9126b031 36cc6fdb c70b8b46 d9e66a48 56e55a79
026a4ceb 52437eff 2f8f76b4 0df980a5 8674cde3 edda04eb 17a9be04 2c18f4df
b7747f9d ab2af7b4 efc34d20 2e096b7c 1741a254 e5b6a035 213d42f6 2c1c7c26
61c2f50f 6552daf9 d2c231f8 25130f69 d8167fa2 0418f2c8 001a96a6 0d1526ab
63315c21 5e0a72ec 49bafefd 187908d9 8d0dbd86 311170a7 3e9b640c cc3e10d7
d5cad3b6 0caec388 f73001e1 6c728aff 71eae2a1 1f9af36e cfcbd12f c1de8417
ac07be6b cb44a1d8 8b9b0f56 013988c3 b1c52fca b4be31cd d8782806 12a3a4e2
6f7de532 58fd7eb6 d01ee900 24adffc2 f4990fc5 9711aac5 001d7b95 82e5e7d2
109873f6 00613096 c32d9521 ada121ff 29908415 7fbb977f af9eb3db 29c9ed2a
5ce2a465 a730f32c d0aa3fe8 8a5cc091 d49e2ce7 0ce454a9 d60acd86 015f1919
77079103 dea03af6 78a8565e dee356df 21f05cbe 8b75e387 b3c50651 b8a5c3ef
d8eeb6d2 e523be77 c2154529 2f69efdf afe67afb f470c4b2 f3e0eb5b d6cc9876
39e4460c 1fda8538 1987832f ca007367 a99144f8 296b299e 492fc295 9266beab
b5676e69 9bd3ddda df7e052f db25701c 1b5e51ee f65324e6 6afce36c 0316cc04
8644213e b7dc59d0 7965291f ccd6fd43 41823979 932bcdf6 b657c34d 4edfd282
7ae5290c 3cb9536b 851e20fe 9833557e 13ecf0b0 d3ffb372 3f85c5c1 0aef7ed2
```

$S_5$：

```
7ec90c04 2c6e74b9 9b0e66df a6337911 b86a7fff 1dd358f5 44dd9d44 1731167f
08fbf1fa e7f511cc d2051b00 735aba00 2ab722d8 386381cb acf6243a 69befd7a
e6a2e77f f0c720cd c4494816 ccf5c180 38851640 15b0a848 e68b18cb 4caadeff
5f480a01 0412b2aa 259814fc 41d0efe2 4e40b48d 248eb6fb 8dba1cfe 41a99b02
1a550a04 ba8f65cb 7251f4e7 95a51725 c106ecd7 97a5980a c539b9aa 4d79fe6a
f2f3f763 68af8040 ed0c9e56 11b4958b e1eb5a88 8709e6b0 d7e07156 4e29fea7
6366e52d 02d1c000 c4ac8e05 9377f571 0c05372a 578535f2 2261be02 d642a0c9
df13a280 74b55bd2 682199c0 d421e5ec 53fb3ce8 c8adedb3 28a87fc9 3d959981
5c1ff900 fe38d399 0c4eff0b 062407ea aa2f4fb1 4fb96976 90c79505 b0a8a774
```

```
ef55a1ff e59ca2c2 a6b62d27 e66a4263 df65001f 0ec50966 dfdd55bc 29de0655
911e739a 17af8975 32c7911c 89f89468 0d01e980 524755f4 03b63cc9 0cc844b2
bcf3f0aa 87ac36e9 e53a7426 01b3d82b 1a9e7449 64ee2d7e cddbb1da 01c94910
b868bf80 0d26f3fd 9342ede7 04a5c284 636737b6 50f5b616 f24766e3 8eca36c1
136e05db fef18391 fb887a37 d6e7f7d4 c7fb7dc9 3063fcdf b6f589de ec2941da
26e46695 b7566419 f654efc5 d08d58b7 48925401 c1bacb7f e5ff550f b6083049
5bb5d0e8 87d72e5a ab6a6ee1 223a66ce c62bf3cd 9e0885f9 68cb3e47 086c010f
a21de820 d18b69de f3f65777 fa02c3f6 407edac3 cbb3d550 1793084d b0d70eba
0ab378d5 d951fb0c ded7da56 4124bbe4 94ca0b56 0f5755d1 e0e1e56e 6184b5be
580a249f 94f74bc0 e327888e 9f7b5561 c3dc0280 05687715 646c6bd7 44904db3
66b4f0a3 c0f1648a 697ed5af 49e92ff6 309e374f 2cb6356a 85808573 4991f840
76f0ae02 083be84d 28421c9a 44489406 736e4cb8 c1092910 8bc95fc6 7d869cf4
134f616f 2e77118d b31b2be1 aa90b472 3ca5d717 7d161bba 9cad9010 af462ba2
9fe459d2 45d34559 d9f2da13 dbc65487 f3e4f94e 176d486f 097c13ea 631da5c7
445f7382 175683f4 cdc66a97 70be0288 b3cdcf72 6e5dd2f3 20936079 459b80a5
be60e2db a9c23101 eba5315c 224e42f2 1c5c1572 f6721b2c 1ad2fff3 8c25404e
324ed72f 4067b7fd 0523138e 5ca3bc78 dc0fd66e 75922283 784d6b17 58ebb16e
44094f85 3f481d87 fcfeae7b 77b5ff76 8c2302bf aaf47556 5f46b02a 2b092801
3d38f5f7 0ca81f36 52af4a8a 66d5e7c0 df3b0874 95055110 1b5ad7a8 f61ed5ad
6cf6e479 20758184 d0cefa65 88f7be58 4a046826 0ff6f8f3 a09c7f70 5346aba0
5ce96c28 e176eda3 6bac307f 376829d2 85360fa9 17e3fe2a 24b79767 f5a96b20
d6cd2595 68ff1ebf 7555442c f19f06be f9e0659a eeb9491d 34010718 bb30cab8
e822fe15 88570983 750e6249 da627e55 5e76ffa8 b1534546 6d47de08 efe9e7d4
```

$S_6$：

```
f6fa8f9d 2cac6ce1 4ca34867 e2337f7c 95db08e7 016843b4 eced5cbc 325553ac
bf9f0960 dfa1e2ed 83f0579d 63ed86b9 1ab6a6b8 de5ebe39 f38ff732 8989b138
33f14961 c01937bd f506c6da e4625e7e a308ea99 4e23e33c 79cbd7cc 48a14367
a3149619 fec94bd5 a114174a eaa01866 a084db2d 09a8486f a888614a 2900af98
01665991 e1992863 c8f30c60 2e78ef3c d0d51932 cf0fec14 f7ca07d2 d0a82072
fd41197e 9305a6b0 e86be3da 74bed3cd 372da53c 4c7f4448 dab5d440 6dba0ec3
083919a7 9fbaeed9 49dbcfb0 4e670c53 5c3d9c01 64bdb941 2c0e636a ba7dd9cd
ea6f7388 e70bc762 35f29adb 5c4cdd8d f0d48d8c b88153e2 08a19866 1ae2eac8
284caf89 aa928223 9334be53 3b3a21bf 16434be3 9aea3906 efe8c36e f890cdd9
80226dae c340a4a3 df7e9c09 a694a807 5b7c5ecc 221db3a6 9a69a02f 68818a54
ceb2296f 53c0843a fe893655 25bfe68a b4628abc cf222ebf 25ac6f48 a9a99387
53bddb65 e76ffbe7 e967fd78 0ba93563 8e342bc1 e8a11be9 4980740d c8087dfc
8de4bf99 a11101a0 7fd37975 da5a26c0 e81f994f 9528cd89 fd339fed b87834bf
5f04456d 22258698 c9c4c83b 2dc156be 4f628daa 57f55ec5 e2220abe d2916ebf
4ec75b95 24f2c3c0 42d15d99 cd0d7fa0 7b6e27ff a8dc8af0 7345c106 f41e232f
35162386 e6ea8926 3333b094 157ec6f2 372b74af 692573e4 e9a9d848 f3160289
3a62ef1d a787e238 f3a5f676 74364853 20951063 4576698d b6fad407 592af950
```

```
36f73523 4cfb6e87 7da4cec0 6c152daa cb0396a8 c50dfe5d fcd707ab 0921c42f
89dff0bb 5fe2be78 448f4f33 754613c9 2b05d08d 48b9d585 dc049441 c8098f9b
7dede786 c39a3373 42410005 6a091751 0ef3c8a6 890072d6 28207682 a9a9f7be
bf32679d d45b5b75 b353fd00 cbb0e358 830f220a 1f8fb214 d372cf08 cc3c4a13
8cf63166 061c87be 88c98f88 6062e397 47cf8e7a b6c85283 3cc2acfb 3fc06976
4e8f0252 64d8314d da3870e3 1e665459 c10908f0 513021a5 6c5b68b7 822f8aa0
3007cd3e 74719eef dc872681 073340d4 7e432fd9 0c5ec241 8809286c f592d891
08a930f6 957ef305 b7fbffbd c266e96f 6fe4ac98 b173ecc0 bc60b42a 953498da
fba1ae12 2d4bd736 0f25faab a4f3fceb e2969123 257f0c3d 9348af49 361400bc
e8816f4a 3814f200 a3f94043 9c7a54c2 bc704f57 da41e7f9 c25ad33a 54f4a084
b17f5505 59357cbe edbd15c8 7f97c5ab ba5ac7b5 b6f6deaf 3a479c3a 5302da25
653d7e6a 54268d49 51a477ea 5017d55b d7d25d88 44136c76 0404a8c8 b8e5a121
b81a928a 60ed5869 97c55b96 eaec991b 29935913 01fdb7f1 088e8dfa 9ab6f6f5
3b4cbf9f 4a5de3ab e6051d35 a0e1d855 d36b4cf1 f544edeb b0e93524 bebb8fbd
a2d762cf 49c92f54 38b5f331 7128a454 48392905 a65b1db8 851c97bd d675cf2f
```

$S_7$：

```
85e04019 332bf567 662dbfff cfc65693 2a8d7f6f ab9bc912 de6008a1 2028da1f
0227bce7 4d642916 18fac300 50f18b82 2cb2cb11 b232e75c 4b3695f2 b28707de
a05fbcf6 cd4181e9 e150210c e24ef1bd b168c381 fde4e789 5c79b0d8 1e8bfd43
4d495001 38be4341 913cee1d 92a79c3f 089766be baeeadf4 1286becf b6eacb19
2660c200 7565bde4 64241f7a 8248dca9 c3b3ad66 28136086 0bd8dfa8 356d1cf2
107789be b3b2e9ce 0502aa8f 0bc0351e 166bf52a eb12ff82 e3486911 d34d7516
4e7b3aff 5f43671b 9cf6e037 4981ac83 334266ce 8c9341b7 d0d854c0 cb3a6c88
47bc2829 4725ba37 a66ad22b 7ad61f1e 0c5cbafa 4437f107 b6e79962 42d2d816
0a961288 e1a5c06e 13749e67 72fc081a b1d139f7 f9583745 cf19df58 bec3f756
c06eba30 07211b24 45c28829 c95e317f bc8ec511 38bc46e9 c6e6fa14 bae8584a
ad4ebc46 468f508b 7829435f f124183b 821dba9f aff60ff4 ea2c4e6d 16e39264
92544a8b 009b4fc3 aba68ced 9ac96f78 06a5b79a b2856e6e 1aec3ca9 be838688
0e0804e9 55f1be56 e7e5363b b3a1f25d f7debb85 61fe033c 16746233 3c034c28
da6d0c74 79aac56c 3ce4e1ad 51f0c802 98f8f35a 1626a49f eed82b29 1d382fe3
0c4fb99a bb325778 3ec6d97b 6e77a6a9 cb658b5c d45230c7 2bd1408b 60c03eb7
b9068d78 a33754f4 f430c87d c8a71302 b96d8c32 ebd4e7be be8b9d2d 7979fb06
e7225308 8b75cf77 11ef8da4 e083c858 8d6b786f 5a6317a6 fa5cf7a0 5dda0033
f28ebfb0 f5b9c310 a0eac280 08b9767a a3d9d2b0 79d34217 021a718d 9ac6336a
2711fd60 438050e3 069908a8 3d7fedc4 826d2bef 4eeb8476 488dcf25 36c9d566
28e74e41 c2610aca 3d49a9cf bae3b9df b65f8de6 92aeaf64 3ac7d5e6 9ea80509
f22b017d a4173f70 dd1e16c3 15e0d7f9 50b1b887 2b9f4fd5 625aba82 6a017962
2ec01b9c 15488aa9 d716e740 40055a2c 93d29a22 e32dbf9a 058745b9 3453dc1e
d699296e 496cff6f 1c9f4986 dfe2ed07 b87242d1 19de7eae 053e561a 15ad6f8c
66626c1c 7154c24c ea082b2a 93eb2939 17dcb0f0 58d4f2ae 9ea294fb 52cf564c
9883fe66 2ec40581 763953c3 01d6692e d3a0c108 a1e7160e e4f2dfa6 693ed285
```

```
74904698 4c2b0edd 4f757656 5d393378 a132234f 3d321c5d c3f5e194 4b269301
c79f022f 3c997e7e 5e4f9504 3ffafbbd 76f7ad0e 296693f4 3d1fce6f c61e45be
d3b5ab34 f72bf9b7 1b0434c0 4e72b567 5592a33d b5229301 cfd2a87f 60aeb767
1814386b 30bcc33d 38a0c07d fd1606f2 c363519b 589dd390 5479f8e6 1cb8d647
97fd61a9 ea7759f4 2d57539d 569a58cf e84e63ad 462e1b78 6580f87e f3817914
91da55f4 40a230f3 d1988f35 b6e318d2 3ffa50bc 3d40f021 c3c0bdae 4958c24c
518f36b2 84b1d370 0fedce83 878ddada f2a279c7 94e01be8 90716f4b 954b8aa3
```

$S_8$：

```
e216300d bbddfffc a7ebdabd 35648095 7789f8b7 e6c1121b 0e241600 052ce8b5
11a9cfb0 e5952f11 ece7990a 9386d174 2a42931c 76e38111 b12def3a 37ddddfc
de9adeb1 0a0cc32c be197029 84a00940 bb243a0f b4d137cf b44e79f0 049eedfd
0b15a15d 480d3168 8bbbde5a 669ded42 c7ece831 3f8f95e7 72df191b 7580330d
94074251 5c7dcdfa abbe6d63 aa402164 b301d40a 02e7d1ca 53571dae 7a3182a2
12a8ddec fdaa335d 176f43e8 71fb46d4 38129022 ce949ad4 b84769ad 965bd862
82f3d055 66fb9767 15b80b4e 1d5b47a0 4cfde06f c28ec4b8 57e8726e 647a78fc
99865d44 608bd593 6c200e03 39dc5ff6 5d0b00a3 ae63aff2 7e8bd632 70108c0c
bbd35049 2998df04 980cf42a 9b6df491 9e7edd53 06918548 58cb7e07 3b74ef2e
522fffb1 d24708cc 1c7e27cd a4eb215b 3cf1d2e2 19b47a38 424f7618 35856039
9d17dee7 27eb35e6 c9aff67b 36baf5b8 09c467cd c18910b1 e11dbf7b 06cd1af8
7170c608 2d5e3354 d4de495a 64c6d006 bcc0c62c 3dd00db3 708f8f34 77d51b42
264f620f 24b8d2bf 15c1b79e 46a52564 f8d7e54e 3e378160 7895cda5 859c15a5
e6459788 c37bc75f db07ba0c 0676a3ab 7f229b1e 31842e7b 24259fd7 f8bef472
835ffcb8 6df4c1f2 96f5b195 fd0af0fc b0fe134c e2506d3d 4f9b12ea f215f225
a223736f 9fb4c428 25d04979 34c713f8 c4618187 ea7a6e98 7cd16efc 1436876c
f1544107 bedeee14 56e9af27 a04aa441 3cf7c899 92ecbae6 dd67016d 151682eb
a842eedf fdba60b4 f1907b75 20e3030f 24d8c29e e139673b efa63fb8 71873054
b6f2cf3b 9f326442 cb15a4cc b01a4504 f1e47d8d 844a1be5 bae7dfdc 42cbda70
cd7dae0a 57e85b7a d53f5af6 20cf4d8c cea4d428 79d130a4 3486ebfb 33d3cddc
77853b53 37effcb5 c5068778 e580b3e6 4e68b8f4 c5c8b37e 0d809ea2 398feb7c
132a4f94 43b7950e 2fee7d1c 223613bd dd06caa2 37df932b c4248289 acf3ebc3
5715f6b7 ef3478dd f267616f c148cbe4 9052815e 5e410fab b48a2465 2eda7fa4
e87b40e4 e98ea084 5889e9e1 efd390fc dd07d35b db485694 38d7e5b2 57720101
730edebc 5b643113 94917e4f 503c2fba 646f1282 7523d24a e0779695 f9c17a8f
7a5b2121 d187b896 29263a4d ba510cdf 81f47c9f ad1163ed ea7b5965 1a00726e
11403092 00da6d77 4a0cdd61 ad1f4603 605bdfb0 9eedc364 22ebe6a8 cee7d28a
a0e736a0 5564a6b9 10853209 c7eb8f37 2de705ca 8951570f df09822b bd691a6c
aa12e4f2 87451c0f e0f6a27a 3ada4819 4cf1764f 0d771c2b 67cdb156 350d8384
5938fa0f 42399ef3 36997b07 0e84093d 4aa93e61 8360d87b 1fa98b0c 1149382c
e97625a5 0614d1b7 0e25244b 0c768347 589e8d82 0d2059d1 a466bb1e f8da0a82
04f19130 ba6e4ec0 99265164 1ee7230d 50b2ad80 eaee6801 8db2a283 ea8bf59e
```

## 1.11　ICE 加密算法

ICE（Information Concealment Engine）是由澳大利亚的 Matthew Kwan 于 1997 年提出的一个标准的 Feistel 分组密码。ICE 加密算法的结构和 DES 加密算法完全相同。DES 加密算法自 1977 年问世以来，已经被广泛用于国际标准，许多专家对它进行了专门攻击研究，它的许多弱点已经逐渐暴露出来，如可以被差分攻击和线性攻击等。ICE 加密算法在层函数 $F$ 的设计中采取了一些新的手段来克服 DES 加密算法的不足，并在运行速度和安全两方面进行了折中考虑。

ICE 加密算法的分组长度为 64 比特，密钥长度为 64 比特，迭代层数为 16。64 比特的输入被平分成 32 比特的左右两部分，右半部分和 60 比特的层密钥作为 $F$ 函数的输入，层函数 $F$ 的输出同左半部分进行模 2 加，然后交换左、右两部分，重复该过程至第 16 层，最后一层左、右两部分不交换。同 DES 加密算法相比，ICE 加密算法只是缺少输入和输出的变换。

ICE 加密算法的 $F$ 函数结构同 DES 加密算法的层函数 $F$ 的结构非常相似。

### 1．$E$ 扩展

将输入的 32 比特从左到右依次记为 $p_{31}$、$p_{30}$、…、$p_1$、$p_0$，按下列规律将 32 比特扩展成 40 比特，从左到右记为 4 个 10 比特的字，即 $E_1$、$E_2$、$E_3$、$E_4$。

$$E_1=p_1p_0p_{31}p_{30}p_{29}p_{28}p_{27}p_{26}p_{25}p_{24}$$
$$E_2=p_{25}p_{24}p_{23}p_{22}p_{21}p_{20}p_{19}p_{18}p_{17}p_{16}$$
$$E_3=p_{17}p_{16}p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8$$
$$E_4=p_9p_8p_7p_6p_5p_4p_3p_2p_1p_0$$

### 2．由密钥控制的置换和加乱

层密钥为 60 比特，记为 3 个 20 比特的字 $SK_1$、$SK_2$、$SK_3$；记 $t_l=(E_1<<10)|E_2$，$t_r=(E_3<<10)|E_4$。$SK_3$ 用于密钥置换，当 $SK_3$ 的某些比特为 1 时，$t_l$ 和 $t_r$ 的相应比特进行交换，可用下面的公式来表示。

$$t=SK_3\&(t_l\oplus t_r)$$
$$t_r=t_r\oplus t$$
$$t_l=t_l\oplus t$$

$SK_1$ 和 $t_l$ 进行模 2 加后作为前两个 $S$ 盒的输入，$SK_2$ 和 $t_r$ 进行模 2 加后作为后两个 $S$ 盒的输入。

### 3．$S$ 盒

ICE 加密算法有 4 个 $S$ 盒，每个 $S$ 盒的输入为 10 比特，输出为 8 比特。同 DES 加密算

法相比，虽然 ICE 加密算法的 $S$ 盒在数量上是减少了，但是每个 $S$ 盒都不是固定不变的，而是依据输入数据的变化而变化的，而且变化量相当大。记 $S$ 盒的 10 比特为 $X$，则 $C$ 为 $X$ 的中间 8 比特组成的十进制数；$R$ 为 $X$ 的最高比特和最低比特合成的十进制数；$O_R$ 为每个 $S$ 盒第 $R$ 行的偏移量；$P_R$ 为每个 $S$ 盒第 $R$ 行上的 Galois 域素数，则每个 $S$ 盒可由公式$(C\text{^}O_R)^7 \bmod P_R$ 生成。ICE 加密算法 $S$ 盒的 $O_R$ 和 $P_R$ 值如表 1.11.1 所示，表中，$O_R$ 的值是十六进制的；$P_R$ 的值是十进制的。

表 1.11.1　ICE 加密算法 $S$ 盒的 $O_R$ 和 $P_R$ 值

| $S$ 盒 | $O_1$ | $O_2$ | $O_3$ | $O_4$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | 83 | 85 | 9b | cd | 333 | 313 | 505 | 369 |
| $S_2$ | cc | a7 | ad | 41 | 379 | 375 | 319 | 391 |
| $S_3$ | 4b | 2e | d4 | 33 | 361 | 445 | 451 | 397 |
| $S_4$ | ea | cd | 2e | 04 | 397 | 425 | 395 | 505 |

### 4．$P$ 置换

4 个 $S$ 盒的输出经一个固定的 $P$ 置换后合成一个新的 32 比特输出，并作为层函数 $F$ 的输出。ICE 加密算法的 $P$ 置换表如表 1.11.2 所示。

表 1.11.2　ICE 加密算法的 $P$ 置换表

| 输出位置 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|---|---|---|---|---|---|---|---|---|
| 原位置 | $S_{1,7}$ | $S_{4,7}$ | $S_{3,7}$ | $S_{2,7}$ | $S_{2,6}$ | $S_{3,6}$ | $S_{1,6}$ | $S_{4,6}$ |
| 输出位置 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 原位置 | $S_{3,5}$ | $S_{2,5}$ | $S_{4,5}$ | $S_{1,5}$ | $S_{4,4}$ | $S_{1,4}$ | $S_{2,4}$ | $S_{3,4}$ |
| 输出位置 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| 原位置 | $S_{2,3}$ | $S_{3,3}$ | $S_{4,3}$ | $S_{1,3}$ | $S_{1,2}$ | $S_{4,2}$ | $S_{2,2}$ | $S_{3,2}$ |
| 输出位置 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 原位置 | $S_{4,1}$ | $S_{1,1}$ | $S_{3,1}$ | $S_{2,1}$ | $S_{3,0}$ | $S_{2,0}$ | $S_{1,0}$ | $S_{4,0}$ |

将 64 比特的密钥从左到右均分成 4 个 16 比特的分组，记为 KB[0]、KB[1]、KB[2]、KB[3]，根据表 1.11.3 所示的密钥移位表从 4 个分组中选出某个分组，取出其低比特，依次合成 16 个 60 比特的层密钥，用 C 语言表示的伪代码如下：

```
for n=1 to 16 do
{
    SK=0;
        for j=1 to 3 do
            for t=1 to 5 do
                for i=0 to 3 do
                {
```

```
                    z=(i+K_R[n])mod 4;
                    B=KB[z]&1;
                    SK=(SK<<1)|B;
                    KB[z]=(KB[z]>>1)|(~B<<19);
                }
        k_n=SK;
    }
```

表 1.11.3　ICE 加密算法的密钥移位表

| 层数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $K_R$ | 0 | 1 | 2 | 3 | 2 | 1 | 3 | 0 | 1 | 3 | 2 | 0 | 3 | 1 | 0 | 2 |

## 1.12　MISTY 算法

MISTY 是日本 Mitsubishi 电气公司的 Mitsuru Matsui 于 1997 年提出的一个分组密码，它采用递归结构，整个结构分成了三个等级，但加密算法的主框架仍采用平衡 Feistel 结构。MISTY 加密算法的分组长度为 64 比特，密钥长度为 128 比特，迭代层数是可变的，但要求必须是 4 的整数倍。MISTY 加密算法采用能有效抵御差分攻击和线性攻击的理论，具有在各种硬件平台（如 ATM 网络）和软件环境（如 IC 卡）中高速运行的能力，是第一个被证明能有效抵御差分攻击和线性攻击的实用分组密码。

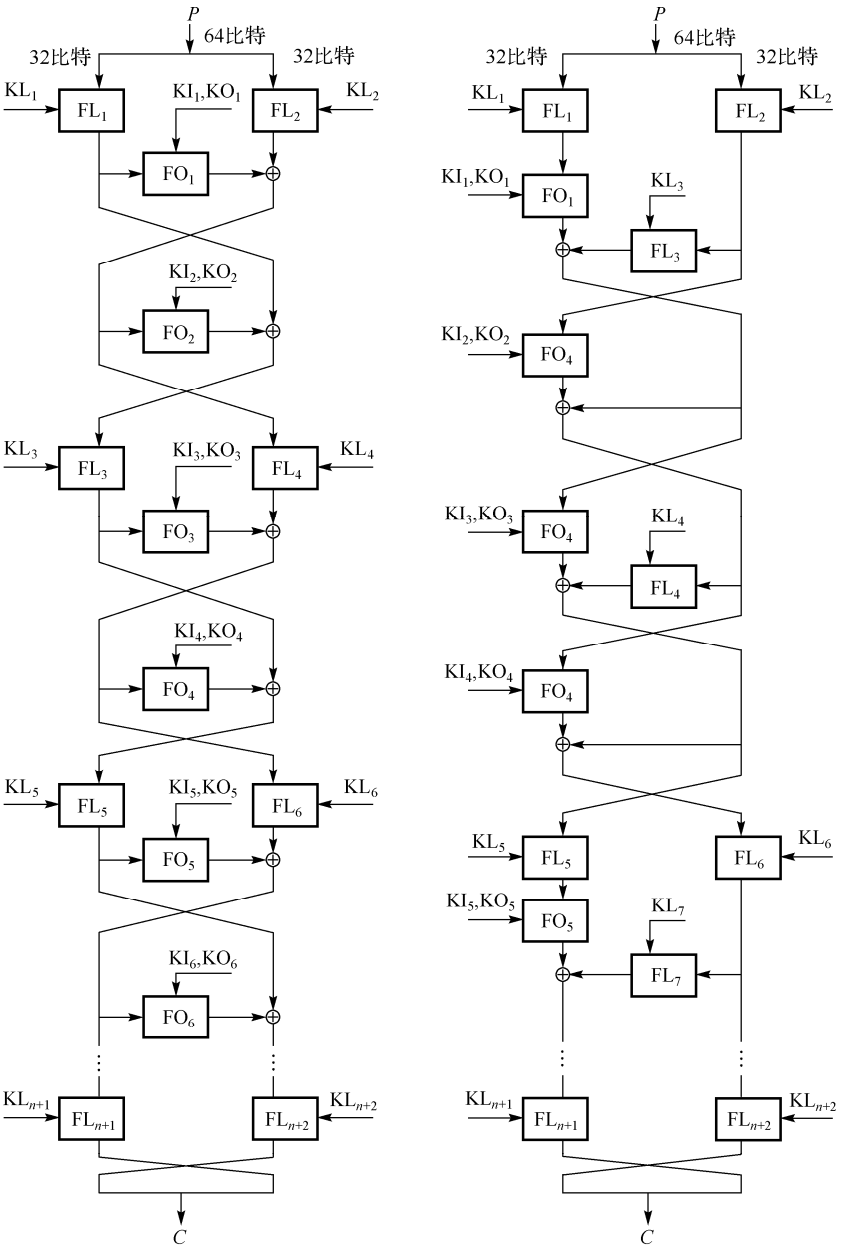
MISTY 加密算法的递归结构如下：第一级是 $n$ 层的主框架，每一层包含加乱（如按位异或操作 $\oplus$、按位或操作 $\cup$、按位与操作 $\cap$）、左右两边异或，以及层函数 FO 变换。第一级的层函数 FO 是一个由加乱（$\oplus$）、左右两边异或和子函数 FI 变换构成的三层迭代过程，这个过程就是第二级。第二级的层函数 FI 又是一个由加乱（$\oplus$）、左右两边扩展和压缩后异或，以及表 $S_9$ 和 $S_7$ 代替构成的，这个过程是第三级。从第一级到第三级，虽然结构逐渐简单，但运算次数比较繁复，每层要加乱 9 次（1 次 $\cup$ 运算、1 次 $\cap$ 运算、7 次 $\oplus$ 运算），代替 9 次（6 次 $S_9$ 代替、3 次 $S_7$ 代替），左右两半进行 16 次 $\oplus$ 运算，但是这种递归结构能通过对简单函数的分析非常容易地估计出算法的安全程度。

MISTY 加密算法有 MISTY1 加密算法和 MISTY2 加密算法两种，它们具有非常相似的结构。对于相同层数的 MISTY1 加密算法和 MISTY2 加密算法，如果不考虑并行运算，则二者在总的算法复杂度上是完全相同的；如果考虑并行计算，则二者的运行时间就有差异，因为 MISTY1 加密算法能同时执行 2 个 FI 函数，而 MISTY2 加密算法能同时执行 4 个 FI 函数。

MISTY1 加密算法和 MISTY2 加密算法的结构如图 1.12.1 所示。

图 1.12.1（a）和图 1.12.1（b）分别给出了 MISTY1 加密算法和 MISTY2 加密算法的结构，在递归结构中的第一级，64 比特的输入被均分成 32 比特的左右两个部分，经过异或运

算，以及层函数 $FO_i$（$1 \leqslant i \leqslant n$）和 $FL_i$（$1 \leqslant i \leqslant n+2$）转换成 64 比特密文。$FO_i$ 使用 64 比特的子密钥 $KO_i$ 和 48 比特的子密钥 $KI_i$，$FL_i$ 使用 32 比特的子密钥 $KL_i$。




（a）MISTY1加密算法　　（b）MISTY2加密算法

图 1.12.1　MISTY1 加密算法和 MISTY2 加密算法的结构

图 1.12.2（a）是层函数 $FO_i$ 的结构，即递归结构中的第二级。32 比特的输入被均分成 16 比特的左右两部分，经过 $\oplus$ 运算，以及层函数 $FI_{ij}$（$1 \leqslant j \leqslant 3$）转换成 32 比特的输出。图中的 $KO_{ij}$（$1 \leqslant j \leqslant 4$）和 $KI_{ij}$（$1 \leqslant j \leqslant 3$）分别是子密钥 $KO_i$ 和 $KI_i$ 的从左边数的第 $j$ 个 16 比特。

图 1.12.2（b）是层函数 $FI_{ij}$ 的结构，即递归结构中的第三级，16 比特的输入被分成 9 比特和 7 比特的左右两部分，经过 $\oplus$ 运算、一个长度为 $2^7$ 的 $S_7$ 代替和一个长度为 $2^9$ 的 $S_9$ 代替后转换成 16 比特的输出。 在第一和第三个 $\oplus$ 运算中，右边的 7 比特在高位添加 2 个“0”扩展成 9 比特，在第二个 $\oplus$ 运算中，右边的 9 比特去掉高位的 2 比特后被截短成 7 比特。图 1.12.2（b）中的 $KI_{ij1}$ 和 $KI_{ij2}$ 分别是子密钥 $KI_{ij}$ 的左边 7 比特和右边 9 比特。

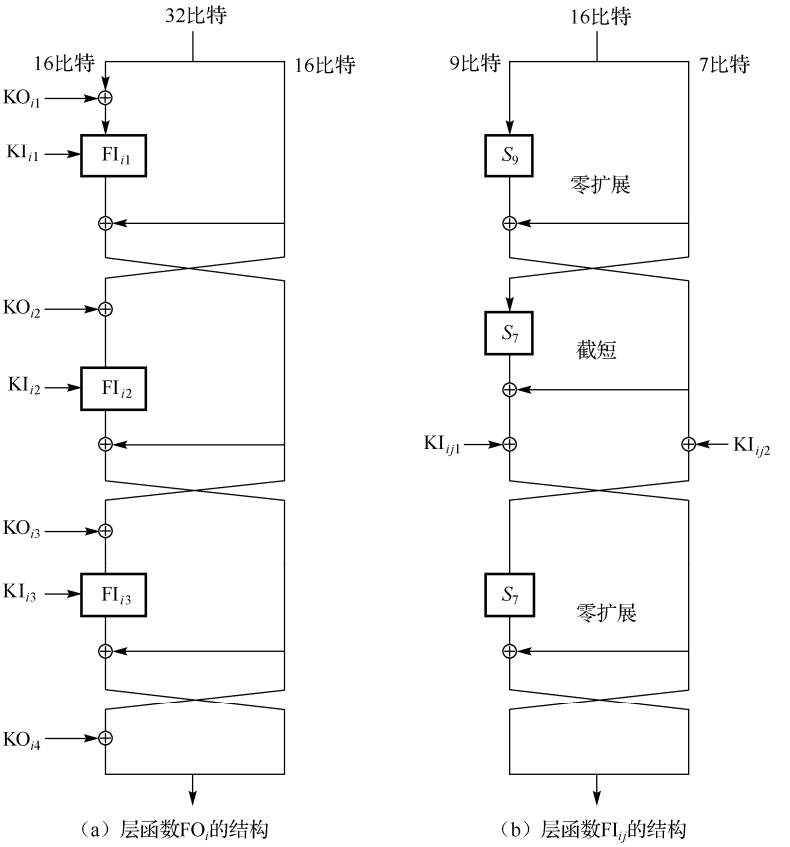



图 1.12.2　MISTY1 加密算法和 MISTY2 加密算法层函数的结构

图 1.12.3 是层函数 $FL_i$ 的结构。32 比特输入被均分成 16 比特的左右两部分，经过 $\oplus$ 操作、与操作（$\cap$）、或操作（$\cup$）后变成 32 比特的输出。$KL_{ij}$（$1 \leqslant j \leqslant 2$）是 $KL_i$ 从左边数第 $j$ 个 16 比特。

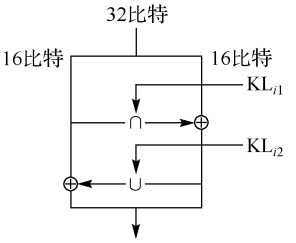



图 1.12.3　层函数 $FL_i$ 结构

MISTY 的子密钥生成过程利用了前面介绍的层函数 FI，记 $K_i$ 为输入密钥 $K$ 的从左边数的第 $i$ 个 16 比特，分别以 $K_i$ 作为层函数 $FI_i$ 的输入，以 $K_{i+1}$ 作为层函数 FI 的密钥，$FI_i$ 的 16 比特输出记为 $R_i$，其中 $1 \leqslant i \leqslant 8$，则子密钥与 $K_i$、$R_i$ 的对应关系如下。

| $KO_{i1}$ | $KO_{i2}$ | $KO_{i3}$ | $KO_{i4}$ | $KI_{i1}$ | $KI_{i2}$ | $KI_{i3}$ | $KL_{i1}$ | $KL_{i2}$ | |
|---|---|---|---|---|---|---|---|---|---|
| $K_i$ | $K_{i+2}$ | $K_{i+7}$ | $K_{i+4}$ | $R_{i+5}$ | $R_{i+1}$ | $R_{i+3}$ | $K_{(i+1)/2}$ | $R_{(i+1)/2+6}$ | （$i$ 为奇数） |
| | | | | | | | $R_{i/2+2}$ | $K_{i/2+4}$ | （$i$ 为偶数） |

上面下角中的加法表示模 8 加。

MISTY 加密算法中使用的 $S_7$ 和 $S_9$ 如下：

$S_7$：

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 27 | 50 | 51 | 90 | 59 | 16 | 23 | 84 | 91 | 26 | 114 | 115 | 107 | 44 | 102 | 73 |
| 31 | 36 | 19 | 108 | 55 | 46 | 63 | 74 | 93 | 15 | 64 | 86 | 37 | 81 | 28 | 4 |
| 11 | 70 | 32 | 13 | 123 | 53 | 68 | 66 | 43 | 30 | 65 | 20 | 75 | 121 | 21 | 111 |
| 14 | 85 | 9 | 54 | 116 | 12 | 103 | 83 | 40 | 10 | 126 | 56 | 2 | 7 | 96 | 41 |
| 25 | 18 | 101 | 47 | 48 | 57 | 8 | 104 | 95 | 120 | 42 | 76 | 100 | 69 | 117 | 61 |
| 89 | 72 | 3 | 87 | 124 | 79 | 98 | 60 | 29 | 33 | 94 | 39 | 106 | 112 | 77 | 58 |
| 1 | 109 | 110 | 99 | 24 | 119 | 35 | 5 | 38 | 118 | 0 | 49 | 45 | 122 | 127 | 97 |
| 80 | 34 | 17 | 6 | 71 | 22 | 82 | 78 | 113 | 62 | 105 | 67 | 52 | 92 | 88 | 125 |

$S_9$：

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 451 | 203 | 339 | 415 | 483 | 233 | 251 | 53 | 385 | 185 | 279 | 491 | 307 | 9 | 45 | 211 |
| 199 | 330 | 55 | 126 | 235 | 356 | 403 | 472 | 163 | 286 | 85 | 44 | 29 | 418 | 355 | 280 |
| 331 | 338 | 466 | 15 | 43 | 48 | 314 | 229 | 273 | 312 | 398 | 99 | 227 | 200 | 500 | 27 |
| 1 | 157 | 248 | 416 | 365 | 499 | 28 | 326 | 125 | 209 | 130 | 490 | 387 | 301 | 244 | 414 |
| 467 | 221 | 482 | 296 | 480 | 236 | 89 | 145 | 17 | 303 | 38 | 220 | 176 | 396 | 271 | 503 |
| 231 | 364 | 182 | 249 | 216 | 337 | 257 | 332 | 259 | 184 | 340 | 299 | 430 | 23 | 113 | 12 |
| 71 | 88 | 127 | 420 | 308 | 297 | 132 | 349 | 413 | 434 | 419 | 72 | 124 | 81 | 458 | 35 |
| 317 | 423 | 357 | 59 | 66 | 218 | 402 | 206 | 193 | 107 | 159 | 497 | 300 | 388 | 250 | 406 |
| 481 | 361 | 381 | 49 | 384 | 266 | 148 | 474 | 390 | 318 | 284 | 96 | 373 | 463 | 103 | 281 |
| 101 | 104 | 153 | 336 | 8 | 7 | 380 | 183 | 36 | 25 | 222 | 295 | 219 | 228 | 425 | 82 |
| 265 | 144 | 412 | 449 | 40 | 435 | 309 | 362 | 374 | 223 | 485 | 392 | 197 | 366 | 478 | 433 |
| 195 | 479 | 54 | 238 | 494 | 240 | 147 | 73 | 154 | 438 | 105 | 129 | 293 | 11 | 94 | 180 |
| 329 | 455 | 372 | 62 | 315 | 439 | 142 | 454 | 174 | 16 | 149 | 495 | 78 | 242 | 509 | 133 |
| 253 | 246 | 160 | 367 | 131 | 138 | 342 | 155 | 316 | 263 | 359 | 152 | 464 | 489 | 3 | 510 |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 189 | 290 | 137 | 210 | 399 | 18 | 51 | 106 | 322 | 237 | 368 | 283 | 226 | 335 | 344 | 305 |
| 327 | 93 | 275 | 461 | 121 | 353 | 421 | 377 | 158 | 436 | 204 | 34 | 306 | 26 | 232 | 4 |
| 391 | 493 | 407 | 57 | 447 | 471 | 39 | 395 | 198 | 156 | 208 | 334 | 108 | 52 | 498 | 110 |
| 202 | 37 | 186 | 401 | 254 | 19 | 262 | 47 | 429 | 370 | 475 | 192 | 267 | 470 | 245 | 492 |
| 269 | 118 | 276 | 427 | 117 | 268 | 484 | 345 | 84 | 287 | 75 | 196 | 446 | 247 | 41 | 164 |
| 14 | 496 | 119 | 77 | 378 | 134 | 139 | 179 | 369 | 191 | 270 | 260 | 151 | 347 | 352 | 360 |
| 215 | 187 | 102 | 462 | 252 | 146 | 453 | 111 | 22 | 74 | 161 | 313 | 175 | 241 | 400 | 10 |
| 426 | 323 | 379 | 86 | 397 | 358 | 212 | 507 | 333 | 404 | 410 | 135 | 504 | 291 | 167 | 440 |
| 321 | 60 | 505 | 320 | 42 | 341 | 282 | 417 | 408 | 213 | 294 | 431 | 97 | 302 | 343 | 476 |
| 114 | 394 | 170 | 150 | 277 | 239 | 69 | 123 | 141 | 325 | 83 | 95 | 376 | 178 | 46 | 32 |
| 469 | 63 | 457 | 487 | 428 | 68 | 56 | 20 | 177 | 363 | 171 | 181 | 90 | 386 | 456 | 468 |
| 24 | 375 | 100 | 207 | 109 | 256 | 409 | 304 | 346 | 5 | 288 | 443 | 445 | 224 | 79 | 214 |
| 319 | 452 | 298 | 21 | 6 | 255 | 411 | 166 | 67 | 136 | 80 | 351 | 488 | 289 | 115 | 382 |
| 188 | 194 | 201 | 371 | 393 | 501 | 116 | 460 | 486 | 424 | 405 | 31 | 65 | 13 | 442 | 50 |
| 61 | 465 | 128 | 168 | 87 | 441 | 354 | 328 | 217 | 261 | 98 | 122 | 33 | 511 | 274 | 264 |
| 448 | 169 | 285 | 432 | 422 | 205 | 243 | 92 | 258 | 91 | 473 | 324 | 502 | 173 | 165 | 58 |
| 459 | 310 | 383 | 70 | 225 | 30 | 477 | 230 | 311 | 506 | 389 | 140 | 143 | 64 | 437 | 190 |
| 120 | 0 | 172 | 272 | 350 | 292 | 2 | 444 | 162 | 234 | 112 | 508 | 278 | 348 | 76 | 450 |

## 1.13　TEA 加密算法

TEA 是一种小型的加密算法，是由英国剑桥大学计算机实验室的 D.J.Wheeler 和 R.M.Needham 于 1994 年提出的。

TEA 是 DES 型多层迭代分组加密算法，迭代层数 $r=2n$，分组长度为 64 比特，密钥长度为 128 比特。TEA 加密算法是一个很简单的程序，设置时间极短，不需要预置非线性代替表。TEA 加密算法的每层变换都很简单，是一个弱非线性变换，经过足够层数的迭代，可以使算法达到安全、保密的目的，所以迭代层数比较多，建议使用 64 层。

128 比特的密钥分成 4 个 32 比特的字，记为 $K_0$～$K_3$，层密钥为 2 个字，第 $i$ 层密钥记为 $K_{i0}$、$K_{i1}$：

$$(K_{i0},K_{i1})=\begin{cases}(K_0,K_1), & \text{当}i\text{为奇数}\\(K_2,K_3), & \text{当}i\text{为偶数}\end{cases}$$

TEA 加密算法还要用到一个常数 $C$，用十六进制表示为：

$$C=9e3779b9=(\sqrt{5}-1)\times 2^{31}$$

$C$ 来自黄金分割数，由 $C$ 生成各层所用常数 $C_1$～$C_r$，即

$$C_{2i-1}=C_{2i}=i\times C \bmod 2^{32},\qquad i=1,\cdots,n$$

输入明文记为（$X_0$,$X_1$），计算

$$X_{i+1}=[(X_i<<4+K_{i0})\oplus(X_i+C_i)\oplus(X_i>>5+K_{i1})]+X_{i-1}$$

式中，“+”为模 $2^{32}$ 加。

TEA 加密算法由交替使用模 2 加和模 $2^{32}$ 加来提供非线性，由相反方向的移位使密钥比特和数据比特反复混合。TEA 加密算法没有设置非线性代替表，使初始设置时间极短。

TEA 加密算法存在等价密钥，当 $K_0$ 和 $K_1$ 的最高位同时取反时等价，同样当 $K_2$ 和 $K_3$ 的最高位同时取反时也等价，所以至少有 3 个等价的密钥。

下面给出 $r$=64 时 TEA 加密算法 C 语言的描述。64 层迭代可以看成 32 个循环，每次循环为 2 层。密钥为 4 个字，放在 k[0]～k[1]，明文放在 k[0]～k[1]。C 语言程序如下：

```
void code (unsigned long*p, unsigned long*k)
{
    unsigned long y=p[0],z=p[1],sum=0,     /*置初值*/
    delta=0x9e3779b9,                      /*常数*/
    n=32;
    while (n>0)
    {
        sum+=delta;
        y+=((z<<4)+k[0])^(z+sum)^((z>>5)+k[1]);
        z+=((y<<4)+k[2])^(y+sum)^((y>>5)+k[3]); n--;
    }
    p[0]=y;p[1]=z;
}
```

经上述运算后，p[0]和 p[1]为密文。

## 1.14 RC5 和 RC6 加密算法

RC5 是 RSA 公钥密码体制的发明者 Ron Rivest 于 1994 年提出的一种分组密码体制，它可看成一种 DES 型的迭代分组密码，以平衡的 Feistel 结构作为算法的基本框架。该算法有三个突出特点：

① 可变性，即分组长度可变、迭代层数可变、密钥长度可变；

② 采用依赖于数据的循环移位；

③ 算法的描述特别简单。

RC5 加密算法将明文和密文分组长度的一半称为字，算法的运算是以字为单位进行的。用 $w$ 表示字的比特数、用 $r$ 表示迭代层数、用 $b$ 表示密钥的字节数，则 RC5-$w$/$r$/$b$ 表示分组长度为 $2w$ 比特、密钥长度为 $b$ 字节的 $r$ 层迭代 RC5 算法。选择不同的参数，可以得到不同的 RC5 加密算法。不同的字长适用于不同字长的处理器，不同的迭代层数可以使用户在速度和安全性两方面进行折中，不同长度的密钥能适应不同安全等级以及美国密码出口控制的要求。

RC6 加密算法是 AES 的候选算法之一，它是 RSA 公司的 Ron Rivest 等人在 RC5 加密算

法的基础上改进而生成的，其分组长度为 128 比特，迭代层数和密钥长度可变。和 RC5 加密算法一样，RC6 加密算法也是用 RC6-$w$/$r$/$b$ 来表示的，推荐的参数为 $w$=32、$r$=20、$b$=16、24 或 32 字节。

### 1.14.1　RC5 加密算法

$b$ 字节的密钥可通过密钥扩展算法得到密钥扩展表，密钥扩展表由 $2r$+2 字，即 $S$[0],$S$[1],…, $S$[$2r$+1]组成。RC5 加密算法对输入的、由两个字组成的明文分组（$A$,$B$）进行以下加密：

$$A=A+S[0]$$
$$B=B+S[1]$$

对于 $i$=1 到 $r$，进行以下运算：

$$A=[(A \oplus B)<<<B]+S[2i]$$
$$B=[(B \oplus A)<<<A]+S[2i+1]$$

式中，“$<<<X$”表示循环左移 $X$ mod $w$ 位。当 $w$ 为 2 的幂次方时，循环移位数为 $X$ 的低 $\log_2 w$ 比特的数值。经上述变化得到的($A$,$B$)即输出的密文。

密钥扩展算法由两个常数，以及三部分变化和计算组成。

两个常数是由自然对数的底数 e=2.718281828459…，以及黄金分割比例 $\phi$=1.618033988749… 来定义的两个字 $P_w$ 和 $Q_w$，即

$$P_w=\text{Odd}[(e-2)2^w]$$
$$Q_w=\text{Odd}[(\phi-1)2^w]$$

式中，Odd($X$)是最接近 $X$ 的奇整数。其实 $P_w$ 和 $Q_w$ 就是由自然对数的底数 e 和黄金分割比例 $\phi$ 小数点后的 $w$ 比特构成的整数。当其为偶数时，加 1 或减 1 后可变成奇数。具体地讲，当小数点后第 $w$+1 比特为 1 时加 1，反之，减 1。$w$=16、32、64 时的常数（用十六进制表示）分别为：

$$P_{16}=\text{b7e1}$$
$$Q_{16}=\text{9e37}$$
$$P_{32}=\text{b7e15163}$$
$$Q_{32}=\text{9e3779b9}$$
$$P_{64}=\text{b7e151628aed2a6b}$$
$$Q_{64}=\text{9e3779b97f4a7c15}$$

密钥扩展表是通过以下三部分变化和计算生成的。

（1）将密钥由字节转换成字。设 $u$=$w$/8，则 $u$ 为每个字的字节数，将 $b$ 字节的密钥 $K$[0],

$K[1],\cdots,K[b-1]$从低比特到高比特排列成字，记为 $L[0],L[1],\cdots,L[c-1]$，其中 $c=[b/u]$。当 $u$ 不能整除 $b$ 时，则在高比特填补 0。

用计算机语言来描述，即先对 $L[0],L[1],\cdots,L[c-1]$置 0，再对 $i=b-1$ 至 0 进行 $L[i/u]=(L[i/u]<<<8)+K[i]$运算。

（2）用常数 $P_w$ 和 $Q_w$ 生成 $S$ 数组。令 $S[0]=P_w$，对于 $i=1$ 到 $2r+1$ 计算 $S[i]=S[i-1]+Q_w \bmod 2^w$，可得 $S$ 数组，即 $S[0],S[1],\cdots,S[2r+1]$。对于给定的字长 $w$，$S$ 数组是固定的以字为单位的常数数组。

（3）将常数数组 $S$ 与密钥数组 $L$ 混乱，生成密钥扩展表。密钥扩展表是由 $S$ 数组和 $L$ 数组经反复累加、移位生成的，累加、移位的次数为两个数组中较长数组字数的 3 倍。具体描述如下：

令 $i=j=0$，$A=B=0$，进行 $3\times\max[(2r+2),c]$次下列运算：

$$A=S[i]=(S[i]+A+B)<<<3$$

$$B=L[j]+(L[j]+A+B)<<<(A+B)$$

$$i=(i+1) \bmod (2r+2)$$

$$j=(j+1) \bmod c$$

计算结束后的 $S[0],S[1],\cdots,S[2r+1]$就是加密所用的密钥扩展表。

### 1.14.2 RC6 加密算法

RC6 加密算法需要 $2r+4$ 个层密钥，由 $b$ 字节的密钥通过密钥扩展算法得到 $2r+4$ 个层密钥的过程同 RC5 加密算法完全一样。

加密的过程如下。

将 128 比特的明文分组分成 4 个 32 比特的字（$A,B,C,D$）后，进行如下运算。

$$B=B+S[0]$$

$$D=D+S[1]$$

对于 $i=1$ 到 $r$，进行如下伪代码所示的运算。

```
{
    t=(B×(2B+1))<<<lgw;
    u=(D×(2D+1))<<<lgw;
    A=((A⊕t)<<<u)+S[2i];
    C=((C⊕u)<<<t)+S[2i+1];
    (A,B,C,D)=(B,C,D,A);
}
A=A+S[2r+2];
C=C+S[2r+3];
```

(*A*,*B*,*C*,*D*)即输出的密文。

## 1.15　RC2 加密算法

RC2 是 Ron Rivest 于 1987 年为 RSA 数据安全公司（RSADSI）设计的专用分组加密算法，它与 RC4 和 RC5 加密算法一起被收入 RSADSI 的加密工具箱 BSAVE。1996 年 2 月有人通过逆性分析，将该加密算法的细节公布在互联网上。1997 年 6 月 RSADSI 正式公布了 RC2 加密算法，并将它提交给国际互联网工程任务组（Internet Engineering Task Force，IETF）作为国际互联网上的标准草案（Internet-Draft）进行评论。RC2 加密算法是 RSADSI 开发的、并已提交 IETF 作为标准审查的电子邮件安全协议 S/MIME 的重要组成部分，RSADSI 公布 RC2 加密算法是为了让 IETF 对它进行审查，这是对 S/MIME 进行标准化工作的一部分。

RC2 加密算法的输入、输出分组长度为 64 比特，输入密钥长度可变（为 1～128 字节）。输入密钥经密钥扩展算法生成 128 字节的扩展密钥 *L*[0]～*L*[127]，作为进行加、脱密运算时的层密钥。RC2 加密算法以 16 比特的字为单位进行运算，输入的 64 比特明文先预置 4 个字 *R*[0]～*R*[3]，然后经两种基本运算——mix（混乱）和 mash（掩乱）多层迭代生成密文，共有 16 层 mix 运算，并在第 5～6 层之间和第 11～12 层 mix 运算之间分别夹有一层 mash 运算。

密钥扩展算法是由输入密钥和 π 表生成 128 字节的层密钥的过程。π 表是 0～255 的排列，是由 π=3.14159…的各位数值决定的。π 表的具体值如表 1.15.1 所示。

表 1.15.1　π 表的具体值

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | d9 | 78 | f9 | c4 | 19 | dd | b5 | ed | 28 | e9 | fd | 79 | 4a | a0 | d8 | 9d |
| 10 | c6 | 7e | 37 | 83 | 2b | 76 | 53 | 8e | 62 | 4c | 64 | 88 | 44 | 8b | fb | a2 |
| 20 | 17 | 9a | 59 | f5 | 87 | b3 | 4f | 13 | 61 | 45 | 6d | 8d | 09 | 81 | 7d | 32 |
| 30 | bd | 8f | 40 | eb | 86 | b7 | 7b | 0b | f0 | 95 | 21 | 22 | 5c | 6b | 4e | 82 |
| 40 | 54 | d6 | 65 | 93 | ce | 60 | b2 | 1c | 73 | 56 | c0 | 14 | a7 | 8c | f1 | dc |
| 50 | 12 | 75 | ca | 1f | 3b | be | e4 | d1 | 42 | 3d | d4 | 30 | a3 | 3c | b6 | 26 |
| 60 | 6f | bf | 0e | da | 46 | 69 | 07 | 57 | 27 | f2 | 1d | 9b | bc | 94 | 43 | 03 |
| 70 | f8 | 11 | c7 | f6 | 90 | ef | 3e | e7 | 06 | c3 | d5 | 2f | c8 | 66 | 1e | d7 |
| 80 | 08 | e8 | ea | de | 80 | 52 | ee | f7 | 84 | aa | 72 | ac | 35 | 4d | 6a | 2a |
| 90 | 96 | 1a | d2 | 71 | 5a | 15 | 49 | 74 | 4b | 9f | d0 | 5e | 04 | 18 | a4 | ec |
| a0 | c2 | e0 | 41 | 6e | 0f | 51 | cb | cc | 24 | 91 | af | 50 | a1 | f4 | 70 | 39 |
| b0 | 99 | 7c | 3a | 85 | 23 | b8 | b4 | 7a | fc | 02 | 36 | 5b | 25 | 55 | 97 | 31 |
| c0 | 2d | 5d | fa | 98 | e3 | 8a | 92 | ae | 05 | df | 29 | 10 | 67 | 6c | ba | c9 |
| d0 | d3 | 00 | e6 | cf | e1 | 9e | a8 | 2c | 63 | 16 | 01 | 3f | 58 | e2 | 89 | a9 |
| e0 | 0d | 38 | 34 | 1b | ab | 33 | ff | b0 | bb | 48 | 0c | 5f | b9 | b1 | cd | 2e |
| f0 | c5 | f3 | db | 47 | e5 | a5 | 9c | 77 | 0a | a6 | 20 | 68 | fe | 7f | c1 | ad |

由表 1.15.1 可知，π(0)=d9，π(1)=78，…，π(ff)=ad。

令 $t$ 为输入密钥的字节数，$s$ 为有效密钥比特数的上界（注意：这里输入密钥长度和有效密钥长度不是一回事，有效密钥长度的比特数为 min(8$t$,$s$)，即穷尽搜索密钥时需要的穷尽量为 $\min(2^{8t},2^{s})$。

令 $t_1$=「$s$/8」为 $s$ 比特对应字节数的上界，$s_1$=($s$ mod 8)+(「($s$+1) /8」−$t_1$)+8，即 $s$ 比特按 8 比特划分余下的比特数，但当 $s$ 能被 8 整除时，$s_1$=8。密钥扩展过程如下：

（1）将输入的 $t$ 字节密钥置入 $L[0],\cdots,L[t-1]$。

（2）对于 $i=t,t+1,\cdots,127$，计算：

$$L[i]=\pi[(L[i-1]+L[i-t]) \bmod 256]$$

（3）计算：

$$L[128-t_1]=\pi[L[128-t_1] \,\&\, (2^{S_1}-1)]$$

式中，“&”表示按比特“与”，该运算以 $L[128-t_1]$的低位 $s_1$ 比特为地址码，取 π 表中的相应值作为新的 $L[128-t_1]$。

（4）对于 $i=127-t_1,\cdots,0$，计算：

$$L[i]=\pi(L[i+1]\oplus L[i+t_1])$$

经过上述四步运算可得层密钥 $L[0]$～$L[127]$，其中步骤（3）和（4）是为了控制有效密钥比特数不超过 $s$，以适应美国政府对出口密码的限制要求。当美国政府限制出口密码的密钥长度不能超过 40 比特时，只要将有效密钥长度的上界 $s$ 设置为 40 即可。即使用户采用很长的密钥（可达 128 字节=1024 比特），但实际有效密钥长度小于或等于 $s$。这是对密码算法设置陷门的方法之一。

RC2 加密算法的加密过程由 mix 和 mash 两个基本运算组成，运算以 16 比特为单位（称其为字）进行。层密钥 $L[0]$～$L[127]$以 16 比特的字作为单位，记为 $K[0]$～$K[63]$，这时 $K[i]=L[2i]+256L[2i+1]$。每层的 mix 用 4 个 $K[i]$作为层密钥，第 1 层的 mix 用 $K[0]$～$K[3]$，第 2 层的 mix 用 $K[4]$～$K[7]$，…，第 $i$ 层的 mix 用 $K[4(i-1)]$～$K[4(i-1)+3]$，16 层共用 64 个 $K[i]$，所以 $K[0]$～$K[63]$中的每个字恰好都被用一次。下面先出 mix 和 mash 的概念，然后介绍一个明文分组的加密过程。

对 $R[i]$进行 mix 运算由以下三步组成：

$$R[i]=R[i]+K[j]+(R[i-1]\,\&\,R[i-2])+(\overline{R[i-1]}\,\&\,R[i-3])$$

$$j=j+1$$

$$R[i]=R[i]<<<r[i]$$

式中，“+”为模 $2^{16}$ 加，“&”为按比特与，$x<<<r[i]$表示 $x$ 循环左移 $r[i]$位，$r[0]$～$r[3]$分别表示 1、2、3、5。$R[i]$中的 $i$ 进行的是模 4 运算，如 $R[-1]=R[3]$。

对 $R[i]$进行 mix 运算，实际上就是在 $R[i]$上加上相应的层密钥 $K[j]$，同时在 $R[i-1]$的指示下选取 $R[i-2]$或 $R[i-3]$的比特合成一个 16 比特的字，即对 $R[i-1]$为“0”的比特、取 $R[i-3]$的相应比特；对 $R[i-1]$为“1”的比特，取 $R[i-2]$的相应比特，组成一个字，加到 $R[i]$上，然后循环左移 $r[i]$位。这相当于一个非平衡的 Feistel 结构。

每层的 mix 运算是对 $i$=0、1、2、3 逐一对 $R[i]$进行 mix 运算。

mash $R[i]$的定义为：

$$R[i]=R[i]+K[R[i-1] \ \& \ 63]$$

即把密钥扩展中的一个字加到 $R[i]$上，以此搅乱 $R[i]$。具体选取密钥扩展中的 $K[0]$～$K[63]$中的哪个字，由 $R[i-1]$的低 6 比特决定，所以 $K[i]$的选取依赖于数据。

每层的 mash 运算是对 $i$=0、1、2、3 逐一进行 mash $R[i]$。

一个明文分组的加密过程如下：

（1）用 64 比特的输入明文分组预置 $R[0]$～$R[3]$；

（2）由输入密钥、π 表以及有效密钥长度的上界 $s$ 进行密钥扩展运算，得到 $K[0]$～$K[63]$；

（3）令 $j$=0；

（4）进行 5 层的 mix 运算；

（5）进行 1 层的 mash 运算；

（6）进行 6 层的 mix 运算；

（7）进行 1 层的 mash 运算；

（8）进行 5 层的 mix 运算。

经上述处理后的 $R[0]$～$R[3]$即密文。

## 1.16　MacGuffin 加密算法

MacGuffin 是由美国的 Matt Blaze 和 Bruce Schneier 在 1994 年剑桥快速软件密码会议上提出的一个分组密码，它采用非平衡 Feistel 结构。MacGuffin 加密算法在分组长度、应用范围和加密框架方面都与 DES 加密算法非常类似，它的输入分组长度为 64 比特，密钥长度为 128 比特，迭代层数为 32 层（从理论上讲，密钥长度和迭代层数可以任意变化）。Matt Blaze 和 Bruce Schneier 设计该算法的主要目的是希望密码分析人员对使用非平衡 Feistel 结构的密码非线性函数的强度进行分析，同时也希望它成为 DES 加密算法的替代品。

算法描述如下：

将 64 比特的输入分成左右不等的两部分，左边 $L_0$ 为 16 比特，右边 $R_0$ 为 48 比特，将右边的 48 个比特从左到右分成 $a$、$b$、$c$ 三个 16 比特的字，与 48 比特的层密钥进行模 2 加，经

过 8 个 $S$ 盒后输出的 16 比特与左边的 16 比特进行模 2 加，模 2 加的结果（16 比特）输入右边的右端，左边移出的 16 比特成为下一层的左边。

该算法的非线性部分是 8 个 $S$ 盒，每个 $S$ 盒的输入为 6 比特，输出为 2 比特。$S$ 盒的输入是从右边 48 比特中固定位置选出的 6 比特。令 $r_{15}$ 代表 $r$ 的最高有效比特，$r_0$ 代表 $r$ 的最低比特。MacGuffin 加密算法的 $S$ 盒如表 1.16.1 所示。

表 1.16.1　MacGuffin 加密算法的 $S$ 盒

| $S$ 盒 | 输入比特 | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| $S_1$ | $a_2$ | $a_5$ | $b_6$ | $b_9$ | $c_{11}$ | $c_{13}$ |
| $S_2$ | $a_1$ | $a_4$ | $b_7$ | $b_{10}$ | $c_8$ | $c_{14}$ |
| $S_3$ | $a_3$ | $a_6$ | $b_8$ | $b_{13}$ | $c_0$ | $c_{15}$ |
| $S_4$ | $a_{12}$ | $a_{14}$ | $b_1$ | $b_2$ | $c_4$ | $c_{10}$ |
| $S_5$ | $a_0$ | $a_{10}$ | $b_3$ | $b_{14}$ | $c_6$ | $c_{12}$ |
| $S_6$ | $a_7$ | $a_8$ | $b_{12}$ | $b_{15}$ | $c_1$ | $c_5$ |
| $S_7$ | $a_9$ | $a_{15}$ | $b_5$ | $b_{11}$ | $c_2$ | $c_7$ |
| $S_8$ | $a_{11}$ | $a_{13}$ | $b_0$ | $b_4$ | $c_3$ | $c_9$ |

从每个 $S$ 盒的 $a$、$b$、$c$ 三部分中各取 2 比特合在一起作为输入，8 个 $S$ 盒各自输出的 2 比特，按表 1.16.2 所示的位置组成一个 16 比特的字 $t$，其中 $t_0$ 为 $t$ 的最低比特，$t_{15}$ 为最高比特。

表 1.16.2　MacGuffin 加密算法 $S$ 盒的输出表

| $S$ 盒 | 输出比特 | |
|---|---|---|
| | 0 | 1 |
| $S_1$ | $t_0$ | $t_1$ |
| $S_2$ | $t_2$ | $t_3$ |
| $S_3$ | $t_4$ | $t_5$ |
| $S_4$ | $t_6$ | $t_7$ |
| $S_5$ | $t_8$ | $t_9$ |
| $S_6$ | $t_{10}$ | $t_{11}$ |
| $S_7$ | $t_{12}$ | $t_{13}$ |
| $S_8$ | $t_{14}$ | $t_{15}$ |

MacGuffin 加密算法的加密过程伪代码如下：

```
left,a,b,c ← I
for i=0 to 31 do
{
    for j=1 to 8 do
    t<= Sj(a⊕K[i,0],b⊕K[i,1],c⊕K[i,2])
```

```
        left ← left⊕t
        left,a,b,c ← a,b,c,left
    }
    O ← left,a,b,c
```

和 DES 加密算法不同，MacGuffin 加密算法的层密钥不是由初始密钥直接移位和抽取得到的，而是以初始密钥为输入，经 MacGuffin 加密算法反复加密，以输出的结果作为层密钥的。32 个层密钥都为 48 比特，将第 $i$ 个层密钥记成 3 个 16 比特字 $K[i,0]$、$K[i,1]$、$K[i,2]$。生成层密钥过程的伪代码如下：

```
K=0
left,a,b,c ← k0...k63
for h=0 to 31 do
{
    for i=0 to 31 do
    {
        for j=1 to 8 do
        t<= Sj(a⊕K[i,0],b⊕K[i,1],c⊕K[i,2])
        left ← left⊕t
        left,a,b,c ← a,b,c,left
    }
    K[h,0] ← left
    K[h,1] ← a
    K[h,2] ← b
}
left,a,b,c ← k64...k127
for h=0 to 31 do
{
    for i=0 to 31 do
    {
        for j=1 to 8 do
        t<= Sj(a⊕K[i,0],b⊕K[i,1],c⊕K[i,2])
        left ← left⊕t
        left,a,b,c ← a,b,c,left
    }
    K[h,0] ← K[h,0]⊕left
    K[h,1] ← K[h,1]⊕a
    K[h,2] ← K[h,2]⊕b
}
```

上述伪代码中，“←”表示赋值；$K[i,j]$表示第 $i$ 个层密钥的第 $j$ 个 16 比特，“<=”表示合并且赋值。

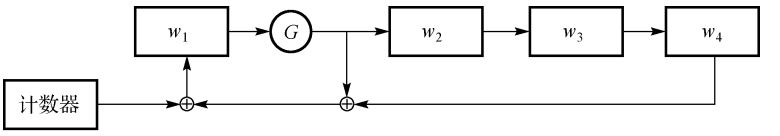
8 个 $S$ 盒如下：

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | *a* | *b* | *c* | *d* | *e* | *f* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_1$ | 0 | 2 | 0 | 0 | 3 | 3 | 1 | 1 | 0 | 0 | 2 | 3 | 0 | 3 | 3 | 2 | 1 |
| | 1 | 1 | 2 | 2 | 0 | 0 | 2 | 2 | 3 | 1 | 3 | 3 | 1 | 0 | 1 | 1 | 2 |
| | 2 | 0 | 3 | 1 | 2 | 2 | 2 | 2 | 0 | 3 | 0 | 0 | 3 | 0 | 1 | 3 | 1 |
| | 3 | 3 | 1 | 2 | 3 | 3 | 1 | 1 | 2 | 1 | 2 | 2 | 0 | 1 | 0 | 0 | 3 |
| $S_2$ | 0 | 3 | 1 | 1 | 3 | 2 | 0 | 2 | 1 | 0 | 3 | 3 | 0 | 1 | 2 | 0 | 2 |
| | 1 | 3 | 2 | 1 | 0 | 0 | 1 | 3 | 2 | 2 | 0 | 0 | 3 | 1 | 3 | 2 | 1 |
| | 2 | 0 | 3 | 2 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 0 | 3 | 3 | 0 | 1 | 0 |
| | 3 | 1 | 3 | 2 | 0 | 2 | 1 | 0 | 2 | 3 | 0 | 1 | 1 | 0 | 2 | 3 | 3 |
| $S_3$ | 0 | 2 | 3 | 0 | 1 | 3 | 0 | 2 | 3 | 0 | 1 | 1 | 0 | 3 | 0 | 1 | 2 |
| | 1 | 1 | 0 | 3 | 2 | 2 | 1 | 1 | 2 | 3 | 2 | 0 | 3 | 0 | 3 | 2 | 1 |
| | 2 | 3 | 1 | 0 | 2 | 0 | 3 | 3 | 0 | 2 | 0 | 3 | 3 | 1 | 2 | 0 | 1 |
| | 3 | 3 | 0 | 1 | 3 | 0 | 2 | 2 | 1 | 1 | 3 | 2 | 1 | 2 | 0 | 1 | 2 |
| $S_4$ | 0 | 1 | 3 | 3 | 2 | 2 | 3 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 0 | 2 | 1 |
| | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 1 | 2 | 2 | 0 | 2 | 3 | 3 |
| | 2 | 2 | 1 | 0 | 3 | 3 | 0 | 0 | 0 | 2 | 2 | 3 | 1 | 1 | 3 | 3 | 2 |
| | 3 | 3 | 3 | 1 | 0 | 1 | 1 | 2 | 3 | 1 | 2 | 0 | 1 | 2 | 0 | 0 | 2 |
| $S_5$ | 0 | 0 | 2 | 2 | 3 | 0 | 0 | 1 | 2 | 1 | 0 | 2 | 1 | 3 | 3 | 0 | 1 |
| | 1 | 2 | 1 | 1 | 0 | 1 | 3 | 3 | 2 | 3 | 1 | 0 | 3 | 2 | 2 | 3 | 0 |
| | 2 | 0 | 3 | 0 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 3 | 2 | 1 | 0 | 2 | 3 |
| | 3 | 3 | 0 | 3 | 3 | 2 | 0 | 1 | 3 | 0 | 2 | 1 | 0 | 0 | 1 | 2 | 1 |
| $S_6$ | 0 | 2 | 2 | 1 | 3 | 2 | 0 | 3 | 0 | 3 | 1 | 0 | 2 | 0 | 3 | 2 | 1 |
| | 1 | 0 | 0 | 3 | 1 | 1 | 3 | 0 | 2 | 2 | 0 | 1 | 3 | 1 | 1 | 3 | 2 |
| | 2 | 3 | 0 | 2 | 1 | 3 | 0 | 1 | 2 | 0 | 3 | 2 | 1 | 2 | 3 | 1 | 2 |
| | 3 | 1 | 3 | 0 | 2 | 0 | 1 | 2 | 1 | 1 | 0 | 3 | 0 | 3 | 2 | 0 | 3 |
| $S_7$ | 0 | 0 | 3 | 3 | 0 | 0 | 3 | 2 | 1 | 3 | 0 | 0 | 3 | 2 | 1 | 3 | 2 |
| | 1 | 1 | 2 | 2 | 1 | 3 | 1 | 1 | 2 | 1 | 0 | 2 | 3 | 0 | 2 | 1 | 0 |
| | 2 | 1 | 0 | 0 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 1 | 0 | 1 | 2 | 2 | 1 |
| | 3 | 2 | 3 | 3 | 1 | 0 | 0 | 2 | 3 | 0 | 2 | 1 | 0 | 3 | 1 | 0 | 2 |
| $S_8$ | 0 | 3 | 1 | 0 | 3 | 2 | 3 | 0 | 2 | 0 | 2 | 3 | 1 | 3 | 1 | 1 | 0 |
| | 1 | 2 | 2 | 3 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 0 | 2 | 2 | 3 | 1 | 0 |
| | 2 | 1 | 0 | 3 | 1 | 0 | 2 | 1 | 1 | 3 | 0 | 2 | 2 | 2 | 2 | 0 | 3 |
| | 3 | 0 | 3 | 0 | 2 | 2 | 3 | 3 | 0 | 3 | 1 | 1 | 1 | 1 | 0 | 2 | 3 |

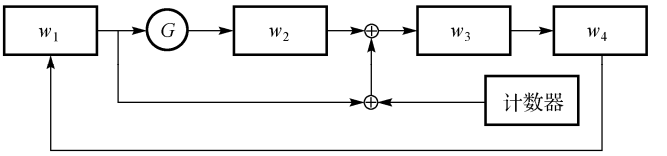
## 1.17 SKIPJACK 加密算法

SKIPJACK 是一种分组加密算法，其输入、输出分组为 64 比特，密钥长度为 80 比特。

输入明文（密文）分成 4 个 16 比特的字，交替使用 A 和 B 两种步进规则，经 32 次步进（即 32 层迭代变换）生成密文（明文）。第 1～8 次步进、17～24 次进步按步进规则 A 进行，第 9～16 次步进、25～32 次步进按步进规则 B 进行。A 和 B 两种步进规则如图 1.17.1 所示。




（a）步进规则A




（b）步进规则B

图 1.17.1　SKIPJACK 算法的两种步进规则

图 1.17.1 中，$w_1$～$w_4$ 表示 4 个 16 比特的字；$G$ 是 $Z_2^{16} \to Z_2^{16}$ 依赖的密钥可逆变换，也称为 $G$ 置换；计数器取值范围 1～32，分别用于 32 次步进。

用上角标 $i$ 表示第 $i$ 步进变化后的状态，即用 $w_1^0 \sim w_4^0$ 表示明文，$w_1^i \sim w_4^i$ 表示经 $i$ 次步进后的状态，则步进规则 A 和 B 可用下列公式来表示。

### 1．加密

（1）步进规则 A 如下：

$$w_1^{i+1} = G^i(w_1^i) \oplus w_4^i \oplus (i+1)$$

$$w_2^{i+1} = G^i(w_1^i)$$

$$w_3^{i+1} = w_2^i$$

$$w_4^{i+1} = w_3^i$$

$$i=0,1,\cdots,31$$

（2）步进规则B如下：

$$w_1^{i+1} = w_4^i$$

$$w_2^{i+1} = G^i(w_1^i)$$

$$w_3^{i+1} = w_1^i \oplus w_2^i \oplus (i+1)$$

$$w_4^{i+1} = w_3^i$$

$$i=0,1,\cdots,31$$

经上述 32 次步进后，可得 $w_1^{32} \sim w_4^{32}$，即密文。

### 2．脱密

（1）步进规则 A 的逆运算：

$$w_1^{i-1} = [G^{i-1}]-1(w_2^i)$$

$$w_2^{i-1} = w_3^i$$

$$w_3^{i-1} = w_4^i$$

$$w_4^{i-1} = w_1^i \oplus w_2^i \oplus i$$

$$i=32,31,\cdots,1$$

（2）进步规则 B 的逆运算：

$$w_1^{i-1} = [G^{i-1}]^{-1}(w_2^i)$$

$$w_2^{i-1} = [G^{i-1}]^{-1}(w_2^i) \oplus w_3^i \oplus i$$

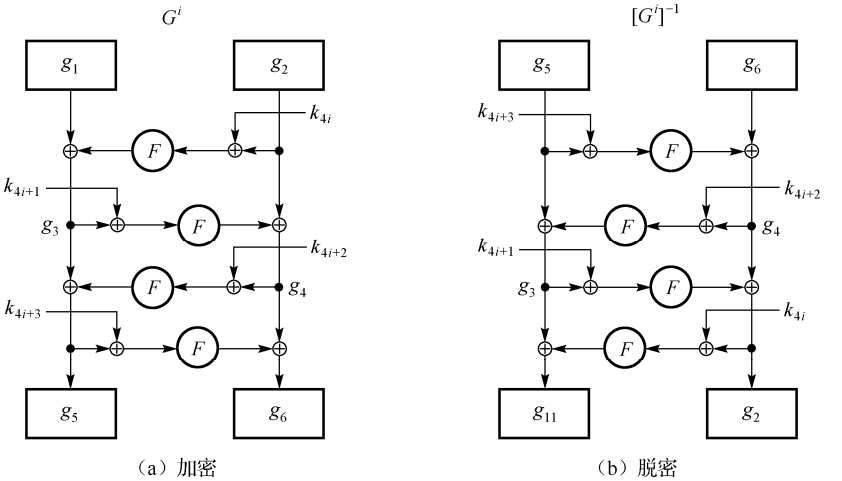
$$w_3^{i-1} = w_4^i$$

$$w_4^{i-1} = w_1^i$$

$$i=32,31,\cdots,1$$

经上述 32 次步进后，可得 $w_1^0 \sim w_4^0$，即明文。

$G$ 置换依赖于密钥的 4 层 Feistel 结构，如图 1.17.2 所示。




（a）加密　　（b）脱密

图 1.17.2　SKIPJACK 加密算法 $G$ 置换

将 80 比特的密钥分成 10 个字节，记为 $k_0$～$k_9$。将第 $i$ 步进输入的 16 比特的字分成高位和低位两个字节，分别记为 $g_1$ 和 $g_2$，迭代计算如下：

$$g_3=F(g_2\oplus k_{4i})\oplus g_1$$
$$g_4=F(g_3\oplus k_{4i+1})\oplus g_2$$
$$g_5=F(g_4\oplus k_{4i+2})\oplus g_3$$
$$g_6=F(g_5\oplus k_{4i+3})\oplus g_4$$

则

$$G^i(g_1\parallel g_2)=g_5\parallel g_6$$

式中，“‖”表示两个字节相连接。在 $k_{4i}$～$k_{4i+3}$ 中，下角进行的是模 10 加。

类似地有

$$(G^i)^{-1}(g_5\parallel g_6)=g_1\parallel g_2$$

式中，$g_j=F(g_{j+1}\oplus k_{4i+j-1})\oplus g_{j+2}$，$j$=4,3,2,1。

$G$ 置换中的 $F$ 函数是 8 比特的字节非线性代替，如下所示。用高 4 比特表示行，低 4 比特表示列，如 $F$(7a)=d6。

| | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xa | xb | xc | xd | xe | xf |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x | a3 | d7 | 09 | 83 | f8 | 48 | f6 | f4 | b3 | 21 | 15 | 78 | 99 | b1 | af | f9 |
| 1x | e7 | 2d | 4d | 8a | ce | 4c | ca | 2e | 52 | 95 | d9 | 1e | 4e | 38 | 44 | 28 |
| 2x | 0a | df | 02 | a0 | 17 | f1 | 60 | 68 | 12 | b7 | 7a | c3 | e9 | fa | 3d | 53 |
| 3x | 96 | 84 | 6b | ba | f2 | 63 | 9a | 19 | 7c | ae | e5 | f5 | f7 | 16 | 6a | a2 |
| 4x | 39 | b6 | 7b | 0f | c1 | 93 | 81 | 1b | ee | b4 | 1a | ea | d0 | 91 | 2f | b8 |
| 5x | 55 | b9 | da | 85 | 3f | 41 | bf | e0 | 5a | 58 | 80 | 5f | 66 | 0b | d8 | 90 |
| 6x | 35 | d5 | c0 | a7 | 33 | 06 | 65 | 69 | 45 | 00 | 94 | 56 | 6d | 98 | 9b | 76 |
| 7x | 97 | fc | b2 | c2 | b0 | fe | db | 20 | e1 | eb | d6 | e4 | dd | 47 | 4a | 1d |
| 8x | 42 | ed | 9e | 6e | 49 | 3c | cd | 43 | 27 | d2 | 07 | d4 | de | c7 | 67 | 18 |
| 9x | 89 | cb | 30 | 1f | 8d | c6 | 8f | aa | c8 | 74 | dc | c9 | 5d | 5c | 31 | a4 |
| ax | 70 | 88 | 61 | 2c | 9f | 0d | 2b | 87 | 50 | 82 | 54 | 64 | 26 | 7d | 03 | 40 |
| bx | 34 | 4b | 1c | 73 | d1 | c4 | fd | 3b | cc | fb | 7f | ab | e6 | 3e | 5b | a5 |
| cx | ad | 04 | 23 | 9c | 14 | 51 | 22 | f0 | 29 | 79 | 71 | 7e | ff | 8c | 0e | e2 |
| dx | 0c | ef | bc | 72 | 75 | 6f | 37 | a1 | ec | d3 | 8e | 62 | 8b | 86 | 10 | e8 |
| ex | 08 | 77 | 11 | be | 92 | 4f | 24 | c5 | 32 | 36 | 9d | cf | f3 | a6 | bb | ac |
| fx | 5e | 6c | a9 | 13 | 57 | 25 | b5 | e3 | bd | a8 | 3a | 01 | 05 | 59 | 2a | 46 |

SKIPJACK 加密算法有类似于 DES 的四种工作方式，它们是：

输出反馈（OFB）方式：64 比特。

密文反馈（CFB）方式：64 比特、32 比特、16 比特、8 比特。

电子密本（ECB）方式：64 比特。

密文分组链接（CBC）方式：64 比特。

例如，电子密本方式（下面所有数值都以十六进制的形式表示，左边为最高比特）。

明文输入为：33221100ddccbbaa。

密钥为：00998877665544332211。

中间步骤如下：

| | $w_1$ | $w_2$ | $w_3$ | $w_4$ |
|---|---|---|---|---|
| 0 | 3322 | 1100 | ddcc | bbaa |
| 1 | b004 | 0baf | 1100 | ddcc |
| 2 | e688 | 3b46 | 0baf | 1100 |
| 3 | 3c76 | 2d75 | 3b46 | 0baf |
| 4 | 4c45 | 47ee | 2d75 | 3b46 |
| 5 | b949 | 820a | 47ee | 2d75 |
| 6 | f0e3 | dd90 | 820a | 47ee |
| 7 | f9b9 | be50 | dd90 | 820a |
| 8 | d79b | 5599 | be50 | dd90 |
| 9 | dd90 | 1e0b | 820b | be50 |
| 10 | be50 | 4c52 | c391 | 820b |
| 11 | 820b | 7f51 | f209 | c391 |
| 12 | c391 | f9c2 | fd56 | f209 |
| 13 | f209 | 25ff | 3a5e | fd56 |
| 14 | fd56 | 65da | d7f8 | 3a5e |
| 15 | 3a5e | 69d9 | 9883 | d7f8 |
| 16 | d7f8 | 8990 | 5397 | 9883 |
| 17 | 9c00 | 0492 | 8990 | 5397 |
| 18 | 9fdc | cc59 | 0492 | 8990 |
| 19 | 3731 | beb2 | cc59 | 0492 |
| 20 | 7afb | 7e7d | beb2 | cc59 |
| 21 | 7759 | bb15 | 7e7d | beb2 |
| 22 | fb64 | 45c0 | bb15 | 7e7d |
| 23 | 6f7f | 1115 | 45c0 | bb15 |
| 24 | 65a7 | deaa | 1115 | 45c0 |
| 25 | 45c0 | e0f9 | bb14 | 1115 |
| 26 | 1115 | 3913 | a523 | bb14 |

| | | | | |
|---|---|---|---|---|
| 27 | bb14 | 8ee6 | 281d | a523 |
| 28 | a523 | bfe2 | 35ee | 281d |
| 29 | 281d | 0d84 | 1adc | 35ee |
| 30 | 35ee | e6f1 | 2587 | 1adc |
| 31 | 1adc | 60ee | d300 | 2587 |
| 32 | 2587 | cae2 | 7a12 | d300 |

密文输出为 2587 cae2 7a12 d300。

## 1.18　BEAR 和 LION 加密算法

BEAR 和 LION 加密算法是由英国的 Ross Anderson 和以色列的 Eli Biham 于 1996 年提出的，是采用 Feistel 结构的分组密码。采用 Feistel 结构的分组密码一般使用多层迭代，每层的层函数由一系列简单的变换组成，而 BEAR 和 LION 加密算法只有三层迭代，分别由一个 Hash 函数或一个序列密码组成。

记分组长度为 $m$ 比特，$H_K(M)$为有密钥加入的 Hash 函数，运算结果长度固定为 $k$ 比特，$S(M)$为序列密码。将 $m$ 比特的明文 $P$ 分成左右不等的两部分 $L$ 和 $R$，$L$ 的长度为 $k$ 比特，$R$ 的长度为$(m-k)$比特。密钥 $K=(K_1,K_2)$，$K_1$ 和 $K_2$ 是两个相互独立的子密钥，长度均大于 $k$。

BEAR 加密算法的加密过程如下：

$$L=L\oplus H_{K_1}(R)$$

$$R=R\oplus S(L)$$

$$L=L\oplus H_{K_2}(R)$$

脱密过程与加密过程相反。

BEAR 加密算法中所用的 Hash 函数 $H_K(M)$需要具备以下三个特点：

① $H_K(M)$是在一般 Hash 函数 $H'(M)$的基础上加入密钥后得到的，即 $H_K(M)=H'(K|M|K)$，符号“|”表示前后两部分相连。

② $H_K(M)$是单向函数，而且是非碰撞的（Collison-Free）。

③ $H_K(M)$是伪随机的，即难以从任一 $H'(X)$推出 $H'(Y)$。

该算法中所用的序列密码 $S(M)$需要具备以下两个特点：

① 能够抵制密钥恢复攻击，即给定 $Y=S(X)$，很难求出 $X$。

② 能够抵制扩展攻击。

LION 加密算法和 BEAR 加密算法的结构几乎完全一致，只是 LION 加密算法使用了一次 Hash 函数，而 BEAR 加密算法使用了两次序列密码。LION 加密算法的加密过程如下：

$$R=R\oplus S(L\oplus K_1)$$
$$L=L\oplus H'(R)$$
$$R=R\oplus S(L\oplus K_2)$$

脱密过程与加密过程相反。

LION 加密算法所用的 Hash 函数要具备的条件可以比 BEAR 加密算法所用的 Hash 函数弱一些，即在 LION 加密算法中，Hash 函数不必是伪随机函数。

由于这两种加密算法的分组长度很大，而对于 Hash 函数来说，输入长度越大，运行速度就越慢。如果使用相同的 Hash 函数和序列密码，LION 加密算法的运行速度要比 BEAR 加密算法快。

上述两种加密算法的安全程度基本取决于所用的 Hash 函数和序列密码。由于两种加密算法的迭代层数较少，在已知密钥的条件下，可以很容易地求出这两个函数，而且这两种加密算法抗选择明/密文攻击的能力也不好。为了增加安全程度，Ross Anderson 和 Eli Biham 又给出了这两种加密算法的一个改进形式——Lioness 加密算法。Lioness 加密算法具有四层结构，抗上述攻击的能力很强。Lioness 加密算法的加密过程如下：

$$R=R\oplus S(L\oplus K_1)$$
$$L=L\oplus H_{K_2}(R)$$
$$R=R\oplus S(L\oplus K_3)$$
$$L=L\oplus H_{K_4}(R)$$

式中，$K_1$、$K_2$、$K_3$、$K_4$ 为 4 个独立的密钥。

脱密过程与加密过程相反。

对于上述三种加密算法，建议使用下面三种工作模式：

① 把整个信息作为一个分组进行加密。

② 可以把信息分成几组，每组使用不同的密钥进行加密。

③ Lioness 加密算法可以使用标准的 DES 模式。

## 1.19 基于伪随机函数和 Hash 函数构造分组密码

### 1.19.1 DES 加密算法

由于 DES 加密算法的公布，使得其 16 轮（Round）乘积密码的框架已被业内人士熟悉了。设其每轮的变换为 $f(\cdot,k_i)$, $i=1,\cdots,16$，$k_i$ 为第 $i$ 轮的子密钥；又设明文分组左半部分的 32 比特为 $L_0$，右半部分的 32 比特为 $R_0$，经 $i$ 轮变换后输出左半部分（32 比特）为 $L_i$，右半部分（32 比特）为 $R_i$, $i=1,\cdots,16$，DES 加密算法的 16 轮乘积密码的框架可简单地表示成（不失

讨论的一般性，输入置换 IP 与输出置换 $IP^{-1}$ 暂不考虑）：

明文为：$(L_0,R_0)$。

第 $i$ 轮输出为：$L_i=R_{i-1}$，$R_i=L_{i-1}\oplus f(R_{i-1},K_i)$，$i=1,\cdots,16$。

密文为：$(R_{16},L_{16})$。

由以上 DES 加密算法容易想到，如果将 DES 加密算法的每轮变换 $f(\cdot,K_i)$换成一个适当的伪随机函数 $F(\cdot,K_i)$，只要 $F$ 的伪随机性好，$K_i$ 足够长，所构成的分组密码的性能也一定良好，并且根据 $F$ 的复杂性，还可以适当地减少整个加密算法的轮数。

### 1.19.2　Phil Karn 加密算法

Phil Karn 曾提出只含两轮的分组密码，轮函数用 Hash 函数 $h$，$h$ 可以是 MD4、MD5 等。

Phil Karn 加密算法可简述如下：

明文为：左半部分为 $L_0$，右半部分为 $R_0$。

加密过程为：　　密文右半部分 $R_1=R_0\oplus h(L_0,K_l)$，$K_l$ 为密钥左半部分　　（1-19-1）

　　密文左半部分 $L_1=L_0\oplus h(R_1,K_r)$，$K_r$ 为密钥右半部分　　（1-19-2）

脱密过程为：已知密文（$L_1, R_1$）和（$K_l, K_r$），由式（1-19-2）可知：

$$L_0=L_1\oplus h(R_1,K_r)$$

由（1-19-1）可知：

$$R_0=R_1\oplus h(L_0,K_l)$$

由此可得到明文$(L_0, R_0)$。

Phil Karn 加密算法的复杂度和安全性完全取决于单向 Hash 函数 $h$ 的强度，但由于迭代的次数过少（只有两轮），在以下特殊情形下，可以由密文分组推出明文分组。

假设已知明文分组为（$P_l, P_r$）及其对应的密文分组为（$C_l, C_r$），还知道另一个明文分组的左半部分与 $P_l$ 相同，若已截收获第二个明文分组的对应密文（$C_l',C_r'$），就可求出第二个明文分组的右半部分 $P_r'$，这是因为已知明文分组（$P_l, P_r$）和对应密文分组（$C_l, C_r$），则式（1-19-1）成为：

$$C_r=P_r\oplus h(P_l, K_l)$$

若 $C_r$、$P_r$ 已知，即可求出 $h(P_l,K_l)$，即

$$h(P_l,K_l)=C_r\oplus P_r \qquad (1\text{-}19\text{-}3)$$

假设第二个明文分组为（$P_l, P_r'$），$P_r'$ 未知，对应的密文分组为（$C_l',C_r'$），若 $C_l'$、$C_r'$ 已知，则式（1-19-1）成为：

$$C_r' = P_r'\oplus h(P_l,K_l)$$

由式（1-19-3）可求出

$$P'_{\mathrm{r}} = C'_{\mathrm{r}} \oplus h(P_{\mathrm{l}}, K_{\mathrm{l}}) = C'_{\mathrm{r}} \oplus C_{\mathrm{r}} \oplus P_{\mathrm{r}}$$

### 1.19.3 Luby-Rackoff 加密算法

Luby-Rackoff 加密算法是在 1988 年提出的 4 轮框架，用以克服上述缺陷，其算法框架如下。

#### 1．加密

明文分组为（$L_0, R_0$），密钥为（$K_{\mathrm{l}}, K_{\mathrm{r}}$），计算：

$$R_1 = R_0 \oplus h(L_0, K_{\mathrm{l}})$$
$$L_1 = L_0 \oplus h(R_1, K_{\mathrm{r}})$$
$$R_2 = R_1 \oplus h(L_1, K_{\mathrm{l}})$$
$$L_2 = L_1 \oplus h(R_2, K_{\mathrm{r}})$$

可得密文分组为（$L_2, R_2$）。

#### 2．脱密

已知密文为（$L_2, R_2$），密钥为（$K_{\mathrm{l}}, K_{\mathrm{r}}$），计算：

$$L_1 = L_2 \oplus h(R_2, K_{\mathrm{r}})$$
$$R_1 = R_2 \oplus h(L_1, K_{\mathrm{l}})$$
$$L_0 = L_1 \oplus h(R_1, K_{\mathrm{r}})$$
$$R_0 = R_1 \oplus h(L_0, K_{\mathrm{l}})$$

可得明文分组为（$L_0, R_0$）。

尽管 Luby-Rackoff 加密算法的安全性未能得到证明，但一般认为它至少与 Hash 函数 $h$ 的安全性相当，甚至要比 DES、IDEA 加密算法的安全性强，并且由于轮数少，因此加密速度很快。

### 1.19.4 Stefan Lucks 加密算法

Stefan Lucks 加密算法在 1994 年提出 3 轮的框架，比 Luby-Rackoff 加密算法速度更快。Stefan Lucks 加密算法用 3 个伪随机函数 $F_1$、$F_2$、$F_3$ 来构成分组密码，其框架如下：

设明文分组为（$L, R$），$L$ 为 $l$ 比特，$R$ 为 $r$ 比特，则：

$$F_1 F_3: [0,1]^r \rightarrow [0,1]^l$$
$$F_2: [0,1]^l \rightarrow [0,1]^r$$

### 1．加密

输入明文分组为（$L, R$），计算：

$$S=L\oplus F_1(R) \tag{1-19-4}$$

$$T=R\oplus F_2(S) \tag{1-19-5}$$

$$U=S\oplus F_3(T) \tag{1-19-6}$$

可得输出的密文分组为（$U, T$）。

### 2．脱密

已知密文分组为（$U, T$），计算：

$$S=U\oplus F_3(T)$$

$$R=T\oplus F_2(S)$$

$$L=S\oplus F_1(R)$$

可得输出的明文分组为（$L, R$）。

Stefan Lucks 加密算法可记为 $F(L,R)=\Psi_{(F_1,F_2,F_3)}(L,R)=(U,T)$。

S.Lucks 证明了只要 $F_1$、$F_2$、$F_3$ 是随机的或伪随机的，则 $F(\cdot,\cdot)$也是随机的或伪随机的。

下面用 Hash 函数来构建以上框架分组密码。假设输入的明文分组为（$L,R$），$L$ 为 $l$ 比特，$R$ 为 $r$ 比特，$r=lb$。$r$ 比特的分组 $R$ 可分成子分组：

$$R=(R_1,\cdots,R_b),\qquad \text{每个 } R_i \text{ 为 } l \text{ 比特}$$

假设输入的明文分组为（$L, R$），加密过程如下：

$$S=L\oplus h(K_1\oplus R)$$

$$T_i=R_i\oplus h[(K_{2,i}\oplus S),\overbrace{K_{2_i},K_{2_i},\cdots,K_{2_i}}^{b-1\text{ 个}}],\qquad i=1,\cdots,b$$

$$U=S\oplus h[K_3\oplus(T_{i_1},\cdots,T_{i_b})],\ \text{某个 } T_{i_0},\ i_0\in[1,\cdots,b]$$

这里举一个 Hash 函数 $h$ 为 MD5 的例子。对于 MD5，有：

$$[0,1]^{512}\rightarrow[0,1]^{128}$$

$K_1$、$K_2$=($K_{2,1}$,$K_{2,2}$,$K_{2,3}$,$K_{2,4}$)和 $K_3$ 都是 512 比特，层密钥 $K_1$、$K_2$、$K_3$ 可以由主密钥 $K$ 经 Hash 函数 MD5（或 RIPEMD）得到，即

$$(K_1,K_2,K_3)=[\mathrm{MD5}(K),\mathrm{MD5}(K+1),\cdots,\mathrm{MD5}(K+11)]$$

或者

$$(K_1,K_2,K_3)=[\mathrm{RIPEMD}(K),\cdots,\mathrm{RIPEMD}(K+11)]$$

明文为 $L$、$R_1$、$R_2$、$R_3$、$R_4$（5×128 比特），加密过程如下：

$$S=L\oplus h[K_1\oplus(R_1,R_2,R_3,R_4)] \tag{1-19-7}$$

$$T_i=R_i\oplus h(K_{2i}\oplus S,K_{2i},K_{2i},K_{2i}),\qquad i=1,2,3,4 \tag{1-19-8}$$

$$U=S\oplus h[K_3\oplus(T_1,T_2,T_3,T_4)] \tag{1-19-9}$$

密文为 $U$、$T_1$、$T_2$、$T_3$、$T_4$。

已知密文 $U$、$T_1$、$T_2$、$T_3$、$T_4$，以及密钥 $K_1$、$K_2$、$K_3$，脱密过程如下：

由式（1-19-9）先求出 $S$，即

$$S=U\oplus h[K_3\oplus(T_1,T_2,T_3,T_4)]$$

由式（1-19-8），可求出 $R_i$，即

$$R_i=T_i\oplus h[(K_{2i}\oplus S),K_{2i},K_{2i},K_{2i}],\qquad i=1,2,3,4$$

由式（1-19-7）求出 $L$，即

$$L=S\oplus h[K_1\oplus(R_1,R_2,R_3,R_4)]$$

得到的明文为 $L$、$R_1$、$R_2$、$R_3$、$R_4$。

上述分组密码的安全性完全取决于 Hash 函数的强度。

基于 Hash 函数来构造分组密码，最引人注目的优点在于：Hash 函数的单向性使得由已知的 Hash 值反求其密钥时，除了穷举所有可能的密钥这一方法，很难能找到其他有效方法。当然这也要看 Hash 函数本身的单向性、伪随机性如何。另外，这种迭代密码（用 Hash 函数作为层函数）可以以较少的轮数获得相同的强度（DES 类型的层函数），因此运算速度会比相同强度的密码算法要快。

Hash 函数可以用 MD4、MD5、RIPEMD 和 SHA，也可以自己设计。

## 1.20 IDEA 加密算法

IDEA 加密算法是来学嘉和他的导师 T. L. Massey 共同提出的，其原型 PES（Proposed Encryption Standard，推荐加密标准）加密算法发表在 1990 年欧洲密码年会上。后来，为了提高算法的抗差值攻击能力，在 1991 年欧洲密码年会上提出了对 PES 加密算法的改进，改进后的加密算法称为 IPES（改进型推荐加密标准），后来正式定名为 IDEA（International Data Encryption Algorithm，国际数据加密算法）。

IDEA 是一种多层迭代分组密码，由 8 层变换和输出变换组成，其输入和输出分组均为 64 比特，分成 4 个子分组，每个子分组为 16 比特；密钥为 128 比特，每层子密钥为 6 个 16 比特，子密钥是按一定方式从 128 比特密钥中选取得到的。

IDEA 加密算法的每层变换由三种互不相容的（Incompatible，即来自不同的代数群）整数通过模运算交替复合而成，其设计的基本思想是把不同代数群中的运算相混乱。

IDEA 加密算法如图 1.20.1 所示。

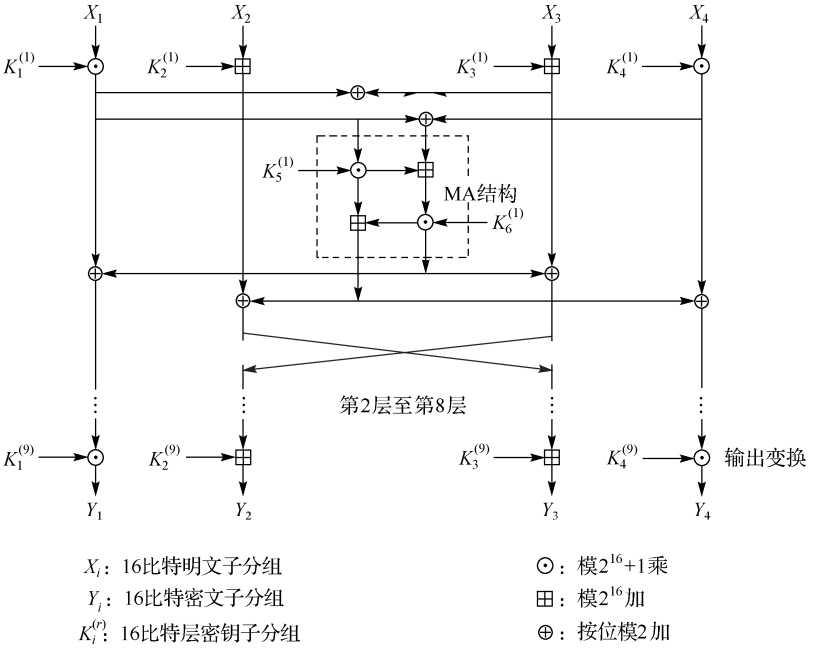



图 1.20.1　IDEA 加密算法

以 16 比特的子分组为单位反复交替进行以下三种不同的群运算：

① 16 比特子分组的对应位进行模 2 加，记为⊕；

② 整数进行模 $2^{16}$ 加，记为⊞，这时可将 16 比特的子分组看成整数的二进制表示；

③ 整数进行模 $2^{16}+1$ 乘，记为⊙，这时也可将 16 比特的子分组看成整数的二进制表示，并把全零子分组看成 $2^{16}$，例如：

$$(0,0,0,\cdots,0)\odot(0,1,0,\cdots,)=(1,1,0,\cdots,0,1)$$

因为 $2^{16}\cdot 2^{14} \bmod (2^{16}+1)=2^{15}+2^{14}+1$。由于 $2^{16}+1$ 是素数，整数模 $2^{16}+1$ 的全体非零元素组成乘法群，因此只要把乘法群中的 $2^{16}$ 与加法群中的零元素相对应，并把整数等同于它的二进制表示，这样上述三种运算就可以看成在同一群上的运算，从而相互交替复合了。

由于这三种运算中的任意两者都不满足分配律和结合律，且一种群中的线性函数相应于另一种群中的非线性函数，此外在层函数的构建中，不使用任一运算的输出作为同样运算的输入，所以密文对明文和密钥的依赖关系复杂度高，达到了混乱（Confusion）的目的。另外，算法中的 MA 结构（Multiplication Addition Structure）使每个输出的密文比特依赖于整个输入明文分组和每个密钥比特，达到了扩散（Diffusion）的目的。

IDEA 的脱密过程与加密过程基本相同，唯一的区别是脱密时的各层子密钥与加密时相比，需进行如下变化。

### 1．加密子密钥

第一层：$K_1^{(1)}\ K_2^{(1)}\ K_3^{(1)}\ K_4^{(1)}\ K_5^{(1)}\ K_6^{(1)}$

第二层：$K_1^{(2)}\ K_2^{(2)}\ K_3^{(2)}\ K_4^{(2)}\ K_5^{(2)}\ K_6^{(2)}$

第三层：$K_1^{(3)}\ K_2^{(3)}\ K_3^{(3)}\ K_4^{(3)}\ K_5^{(3)}\ K_6^{(3)}$

第四层：$K_1^{(4)}\ K_2^{(4)}\ K_3^{(4)}\ K_4^{(4)}\ K_5^{(4)}\ K_6^{(4)}$

第五层：$K_1^{(5)}\ K_2^{(5)}\ K_3^{(5)}\ K_4^{(5)}\ K_5^{(5)}\ K_6^{(5)}$

第六层：$K_1^{(6)}\ K_2^{(6)}\ K_3^{(6)}\ K_4^{(6)}\ K_5^{(6)}\ K_6^{(6)}$

第七层：$K_1^{(7)}\ K_2^{(7)}\ K_3^{(7)}\ K_4^{(7)}\ K_5^{(7)}\ K_6^{(7)}$

第八层：$K_1^{(8)}\ K_2^{(8)}\ K_3^{(8)}\ K_4^{(8)}\ K_5^{(8)}\ K_6^{(8)}$

输出变换：$K_1^{(9)}\ K_2^{(9)}\ K_3^{(9)}\ K_4^{(9)}$

### 2．脱密子密钥

第一层：$K_1^{(9)^{-1}}\ -K_2^{(9)}\ -K_3^{(9)}\ K_4^{(9)^{-1}}\ K_5^{(8)}\ K_6^{(8)}$

第二层：$K_1^{(8)^{-1}}\ -K_2^{(8)}\ -K_3^{(8)}\ K_4^{(8)^{-1}}\ K_5^{(7)}\ K_6^{(7)}$

第三层：$K_1^{(7)^{-1}}\ -K_2^{(7)}\ -K_3^{(7)}\ K_4^{(7)^{-1}}\ K_5^{(6)}\ K_6^{(6)}$

第四层：$K_1^{(6)^{-1}}\ -K_2^{(6)}\ -K_3^{(6)}\ K_4^{(6)^{-1}}\ K_5^{(5)}\ K_6^{(5)}$

第五层：$K_1^{(5)^{-1}}\ -K_2^{(5)}\ -K_3^{(5)}\ K_4^{(5)^{-1}}\ K_5^{(4)}\ K_6^{(4)}$

第六层：$K_1^{(4)^{-1}}\ -K_2^{(4)}\ -K_3^{(4)}\ K_4^{(4)^{-1}}\ K_5^{(3)}\ K_6^{(3)}$

第七层：$K_1^{(3)^{-1}}\ -K_2^{(3)}\ -K_3^{(3)}\ K_4^{(3)^{-1}}\ K_5^{(2)}\ K_6^{(2)}$

第八层：$K_1^{(2)^{-1}}\ -K_2^{(2)}\ -K_3^{(2)}\ K_4^{(2)^{-1}}\ K_5^{(1)}\ K_6^{(1)}$

输出变换：$K_1^{(1)^{-1}}\ -K_2^{(1)}\ -K_3^{(1)}\ K_4^{(1)^{-1}}$

密码变换的自逆特点来自各层中的对合结构，如图 1.20.2 所示。

因为

$$(S_1,S_2)\oplus(S_3,S_4)=(R_1,R_2)\oplus(R_3,R_4)$$

所以当 $S_1$、$S_2$、$S_3$、$S_4$ 分别由 $R_1$、$R_2$、$R_3$、$R_4$ 代替时，MA 结构的输入不变，因此每层的半部分输出为：

$$\begin{aligned}&(R_3,R_4)\oplus \mathrm{MA}[(R_1,R_2)\oplus(R_3,R_4),K_5,K_6]\\=&(S_1,S_2)\oplus \mathrm{MA}[(S_1,S_2)\oplus(S_3,S_4),K_5,K_6)]\oplus \mathrm{MA}[(R_1,R_2)\oplus(R_3,R_4),K_5,K_6]\\=&(S_1,S_2)\end{aligned}$$

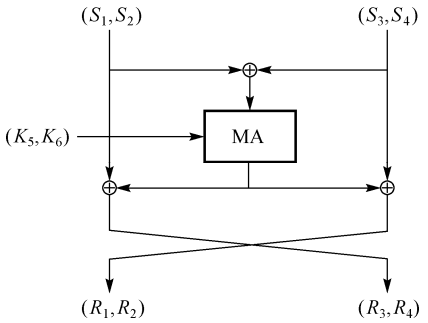



图 1.20.2　IDEA 加密算法中的对合结构

同理，右半部分输出为（$S_3, S_4$），所以 IDEA 加密算法具有自逆的特点。

8 层变换和输出变换所需要的 52 个 16 比特子密钥，是从 128 比特的密钥中按下列方法选取的：把 128 比特的密钥头尾相接，然后从头连续取 8 个 16 比特作为 $K_1^{(1)}, K_2^{(1)}, \cdots, K_6^{(1)}, K_1^{(2)}, K_2^{(2)}$，接着跳过 25 比特再连续取 8 个 16 比特作为 $K_3^{(2)}, K_4^{(2)}, \cdots, K_6^{(2)}, K_1^{(3)}, \cdots, K_4^{(3)}$，如此继续，直到取满 52 个 16 比特的子密钥为止。

## 1.21　REDOC-Ⅱ和 REDOC-Ⅲ加密算法

REDOC-Ⅱ加密算法是由美国 Cryptech 有限公司的 M. C. Wood 和纽约州立大学的 T. W. Cusick 在 1990 年美洲密码年会上发表的，采用非平衡 Feistel 结构，分组长度为 10 字节，密钥长度为 160 比特，迭代层数为 10。REDOC-Ⅱ加密算法的层函数由一系列的置换、代替、与子密钥结合，以及 Enclave 函数等组成，它以字节为单位进行运算，因而加密的速度极快。后来 M. C. Wood 又给出了 REDOC-Ⅱ加密算法的简化型，称为 REDOC-Ⅲ，它的分组长度和迭代层数和 REDOC-Ⅱ加密算法一样，但密钥长度可变，最长可达 2560 字节。REDOC-Ⅲ加密算法只使用了密钥字节和报文之间的模 2 加运算，因此运算速度更快。特别要注意的是，REDOC-Ⅱ加密算法只对每个字节的低位 7 比特加密，因此它适合明文是 ASCII 码的情形。

### 1.21.1　REDOC-Ⅱ加密算法

REDOC-Ⅱ加密算法使用了掩码表、密钥表、置换表、Enclave 函数表和一张代替表，其中，置换表、Enclave 函数表和代替表是预先生产好的，掩码表、密钥表是由输入密钥按一定规律临时生成的。

掩码表由 4 组 10 字节的数组成，4 组掩码分别用 $\text{Mask}_1$、$\text{Mask}_2$、$\text{Mask}_3$ 和 $\text{Mask}_4$ 表示，如表 1.21.1 所示。

表 1.21.1　REDOC-II 加密算法的掩码表

| 字节 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $Mask_1$ | 48 | 2 | 121 | 18 | 60 | 105 | 33 | 50 | 11 | 60 |
| $Mask_2$ | 26 | 78 | 74 | 72 | 69 | 13 | 77 | 43 | 9 | 99 |
| $Mask_3$ | 64 | 113 | 72 | 61 | 37 | 13 | 49 | 71 | 24 | 60 |
| $Mask_4$ | 104 | 62 | 69 | 87 | 18 | 31 | 102 | 101 | 32 | 125 |

4 组掩码分别是由密钥表中的部分子密钥通过模 2 加生成的，密钥表有 128 个子密钥，分别用 $K_i$（$1 \leqslant i \leqslant 128$）来表示，每个子密钥也是由 10 字节组成，密钥表的形状和掩码表相同。掩码表是由密钥表按下式生成的：

$$Mask_i = K_{(i-1)\times 32+1} \oplus K_{(i-1)\times 32+2} \oplus \cdots \oplus K_{(i-1)\times 32+32}, \qquad i=1,2,3,4$$

掩码表中的 4 组掩码分别用来选择加密用的置换表、密钥表、Enclave 函数表和代替表。每组掩码中的第 $i$ 个字节（$1 \leqslant i \leqslant 10$）的掩码与第 $i$ 层的加密相对应。

置换表由 128 个 10 阶置换组成，当每次加密时，从置换表中选出一个置换，对输入的 10 字节进行位置置换。REDOC-Ⅱ加密算法的置换表如表 1.21.2 所示。

表 1.21.2　REDOC-II 加密算法的置换表

| 输入分组 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 置换 0 | 1 | 6 | 7 | 9 | 10 | 2 | 5 | 8 | 3 | 4 |
| 置换 1 | 10 | 4 | 8 | 3 | 1 | 7 | 2 | 9 | 5 | 6 |
| 置换 2 | 9 | 8 | 3 | 4 | 5 | 10 | 6 | 1 | 7 | 2 |
| … | …… | | | | | | | | | |
| 置换 86 | 9 | 7 | 2 | 6 | 5 | 8 | 3 | 10 | 1 | 4 |
| 置换 87 | 5 | 3 | 8 | 1 | 9 | 7 | 10 | 2 | 4 | 6 |
| … | …… | | | | | | | | | |
| 置换 127 | 7 | 8 | 5 | 10 | 9 | 3 | 4 | 2 | 1 | 6 |

Enclave 函数是 REDOC-Ⅱ加密算法独有的一个函数，共有 32 张 Enclave 函数表，每张表由 $a$、$b$、$c$、$d$ 四张子表组成，每张子表有 5 组数，每组数有 3 个值。每张子表都确定了一种对半个分组（5 字节）进行变换的方法。子表 $a$ 和子表 $b$ 用于变换右半分组（部分），子表 $c$ 和子表 $d$ 用于变换左半分组。REDOC-Ⅱ加密算法的 Enclave 函数表如表 1.21.3 所示。

表 1.21.3　REDOC-II 加密算法的 Enclave 函数表

| 组数 | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| 1 | 5 2 3 | 3 5 2 | 5 4 2 | 5 4 2 |
| 2 | 4 3 1 | 1 3 5 | 4 3 1 | 2 5 1 |
| 3 | 2 5 4 | 2 4 1 | 1 5 3 | 1 3 5 |
| 4 | 1 4 5 | 5 1 4 | 3 2 5 | 3 2 4 |
| 5 | 3 1 2 | 4 2 3 | 2 1 4 | 4 1 3 |

REDOC-Ⅱ加密算法有 16 张代替表，每一张代替表由 128 字节组成，用以替换输入的每一个字节值，如输入的第一个字节值为 $i$，则经过代替后的第一个字节值变为代替表中的第 $i$ 个字节值。每次加密时只用其中的 2 张代替表。

由于第 $i$（$1 \leqslant i \leqslant 10$）层加密只与每组掩码中的第 $i$ 个字节有关，因此不妨以第 $i$ 层加密为例进行说明。记第 $i$ 层的输入分组为 $A$，加密过程如下：

（1）对输入分组 $A$ 进行置换。求出 $A$ 的 10 个字节之和模 128 的值 sum($A$)，计算 $W$=sum($A$)$\oplus \text{Mask}_{1,i}$（$\text{Mask}_{j,k}$ 表示第 $j$ 组掩码的第 $k$ 个值），从置换表中选出第 $W$ 个置换，对 $A$ 的 10 字节位置进行置换，记置换后的分组为 $B$。

（2）分组 $B$ 与子密钥进行两次结合。将分组 $B$ 的第 $i$ 个字节记为 $B_i$，计算 $W=B_i \oplus \text{Mask}_{2,i}$，从密钥表中选出第 $W$ 个密钥，将该密钥同分组 $B$ 中的除第 $i$ 个字节以外的每一个字节进行模 2 加，得到分组 $C$。计算 $W=C_x \oplus \text{Mask}_{2,i}$，其中 $x=i+1 \bmod 10$，从密钥表中选出第 $W$ 个密钥，将该密钥同分组 $C$ 中的除第 $x$ 个字节以外的每一个字节进行模 2 加，得到分组 $D$。

（3）用 Enclave 函数对分组 $D$ 进行变换。计算 $W=(D_i \oplus \text{Mask}_{3,i}) \bmod 32$，从 Enclave 函数表中选出第 $W$ 个 Enclave 函数对分组 $D$ 进行变换可得到分组 $E$，具体的变换方法如下：

将分组 $D$ 分成左右两部分，记成 $D_L=(D_1D_2D_3D_4D_5)$和 $D_R=(D_6D_7D_8D_9D_{10})$，首先用子表 $a$ 变换 $D_R$。记子表 $a$ 第一组的三个值为 $l$、$k$、$j$，计算：

$$W=(D_{l+5}+D_{k+5}+D_{j+5}) \bmod 128$$

$$D_{l+5}=W$$

依次用子表 $a$ 的第 1、2、3、4、5 组数，采用上面的方法对 $D_R$ 进行变换，得到新的 $D_R$，记为 $D_{R1}$；再用子表 $b$ 对 $D_{R1}$ 进行与子表 $a$ 同样的变换方法得到 $D_{R2}$；计算 $D_L=D_L \oplus D_{R2}$，依次用子表 $c$ 和子表 $d$ 对 $D_L$ 进行变换（采用与子表 $a$ 相同的方法），得到 $D_{L2}$；计算 $D_{R3}=D_{L2} \oplus D_{R2}$；$E=D_{L2} \| D_{R3}$，即对分组 $D$ 的变换结果。

（4）对分组 $E$ 进行两次代替。计算 $W=(E_i \oplus \text{Mask}_{4,i}) \bmod 16$，从 16 张代替表中选出第 $W$ 张代替表，替换分组 $E$ 中除第 $i$ 个字节以外的每一个字节，得到分组 $F$。计算 $W=(F_x \oplus \text{Mask}_{4,i}) \bmod 16$，$x=i+1 \bmod 10$，从 16 张代替表中选出第 $W$ 张代替表，替换分组 $F$ 中除第 $x$ 个字节外的每一个字节，得到分组 $G$。

分组 $G$ 作为第 $i$ 层的输出，也是第 $i+1$ 层的输入，如果 $i$=10，则 $G$ 就是密文。

### 1.21.2　REDOC-Ⅲ加密算法

（1）利用私有密钥生成一张由 256 个密钥组成的密钥表，其中每个子密钥都是 10 字节。

（2）生成 2 个 10 字节的掩码分组 $\text{Mask}_1$ 和 $\text{Mask}_2$。$\text{Mask}_1$ 为前 128 个密钥模 2 加的结果，$\text{Mask}_2$ 为后 128 个密钥模 2 加的结果。

（3）对一个 10 字节的明文分组 $A$ 进行加密，伪代码如下：

```
    for(i=1; i<=10; i++)
    {
        W=A_i ⊕ Mask_{1,i}
        从密钥表中选出 1 个密钥,将该密钥和数据分组 A 中除第 i 个字节以外的每个字节进行模 2
加，得到新的数据分组 A
    }
```

将上述伪代码中的 $Mask_1$ 换成 $Mask_2$ 即可。

输出的结果就是密文。

## 1.22 SAFERK-64 和 SAFER+加密算法

SAFERK-64 加密算法是 1993 年由瑞士的 James Massey 为 Cylink 公司设计的，它的输入、输出分组长度，以及密钥、层密钥长度均为 64 比特（8 字节）。SAFERK-64 加密算法不是 Feistel 结构，它把输入分组分成 8 字节，以字节为单位对整块同时进行加/脱密；同时混用模 2 加和模 256 加，用 45 mod 257 的幂运算和对数运算代替非线性 $S$ 盒，采用了一个非传统的线性置换——伪 Hadamard 变换来完成信息的充分扩散，还采用了密钥偏移量来减少弱密钥的存在，因而具有极高的安全性。SAFERK-64 的加/脱密过程不完全一致。

SAFER+加密算法是 SAFER 系列密码之一，也是 AES 的候选加密算法之一，其分组长度为 128 比特，密钥长度可以为 128、192、256 比特。对于这三种密钥长度，迭代层数分别为 8、12、16 层。SAFER+加密算法的结构和 SAFERK-64 加密算法完全一样，它只是将 SAFERK-64 加密算法的伪 Hadamard 变换用一个可逆矩阵来代替。下面分别介绍这两种加密算法。

### 1.22.1 SAFERK-64 加密算法

明文分组经过 $r$ 层同样的迭代过程后，再经过一次输出变换便可得到密文。每层需要一对层密钥，输出变换需要一个层密钥，因而整个加密过程需要 $2r+1$ 个层密钥。

SAFERK-64 加密算法的加密过程层函数结构如图 1.22.1 所示，从图中可以看出，第 $i$ 层输入的 8 字节首先与层密钥 $K_{2i-1}$ 的 8 字节对应进行 xor 或 add 运算（其中 xor 代表模 2 加运算，add 表示模 256 加运算），与层密钥结合后输出的 8 字节并进入非线性层。

非线性层由两个函数组成，在图中分别用 exp 和 log 表示，exp 表示如果输入字节的整数值为 $j$，则输出字节为 $45^j \bmod 257$，只有在 $j$=128 时，输出字节为 0，而非 256；log 表示如果输入字节的整数值为 $j$，则输出字节为 $\log_{45}^{(j)}$，只有在 $j$=0 时，输出字节为 128。由于 257

是素数，因此 exp 和 log 表示的运算是有限域 GF(257)中的算术运算，而 45 是这个域中的本原元，它的前 256 次幂就是该域的 256 个非零元，因此，exp 和 log 对于有限域 GF(257)上的运算是一对字节到字节的可逆非线性变换，而且非线性度非常高。

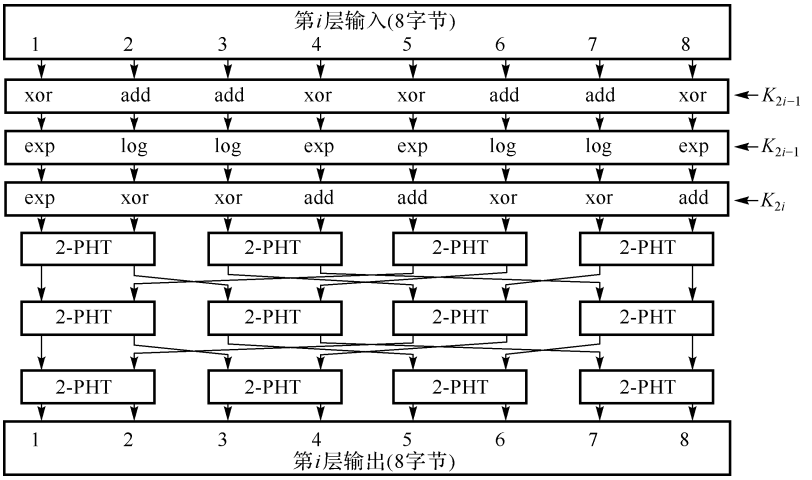



图 1.22.1　SAFERK-64 加密算法的加密过程层函数结构

8 字节分别经过非线性层后，又与层密钥 $K_{2i}$ 的对应字节分别进行 add 或 xor 运算，完成与层密钥的结合后，8 字节进入三级线性层。

三级线性层在图 1.22.1 中是由 12 个符号 2-PHT 来表示的，2-PHT 表示 1 个两头的伪 Hadamard 变换，如果该变换的两个输入字节为（$a_1$,$a_2$），则两个输出字节（$b_1$,$b_2$）为：

$$b_1=2a_1+a_2$$

$$b_2=a_1+a_2$$

式中，“+”表示模 256 加。该层完成了算法所需求的充分扩散功能，至此，层函数完全结束。

输出变换过程是第 $r$ 层输出与层密钥 $K_{2r+1}$ 的结合过程，输入的第 1、4、5、8 字节与 $K_{2r+1}$ 的相应字节进行模 2 加，输入的第 2、3、6、7 字节与 $K_{2r+1}$ 的相应字节进行模 256 加。

SAFERK-64 加密算法的脱密过程同加密过程不完全相同。脱密过程是密文先经过一个输入变换，然后经过 $r$ 层同样的迭代过程后输出明文。

输入变换是加密过程中输出变换的逆变换，输入的第 1、4、5、8 字节与层密钥 $K_{2r+1}$ 的相应字节进行模 2 加，输入的第 2、3、6、7 字节与 $K_{2r+1}$ 的相应字节进行模 256 减（sub）。

SAFERK-64 加密算法的脱密过程层函数结构如图 1.22.2 所示。

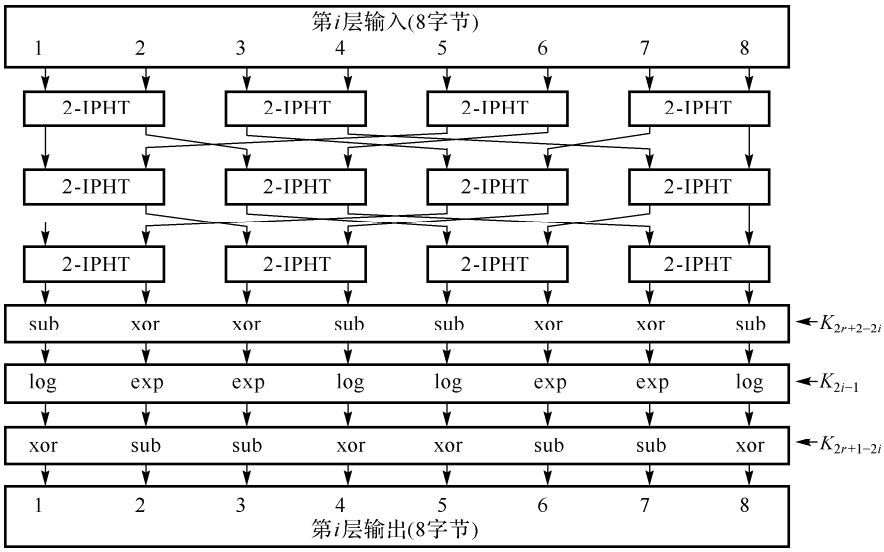



图 1.22.2　SAFERK-64 加密算法的脱密过程层函数结构

图中的 2-IPHT 是加密过程中 2-PHT 的逆变换，如果 2-IPHT 变换中的两个输入字节为（$b_1$,$b_2$），则两个输出字节（$a_1$,$a_2$）为：

$$a_1=b_1-b_2$$
$$a_2=-b_1+2b_2$$

式中，“+”表示模 256 加。

SAFERK-64 加密算法的层密钥生成规律的伪代码如下所示。

```
K1=K; r1=K1;
for(i=2; i<=2r+1; i++)
{
    ri=lrot(ri-1,3);
    Ki=addb(ri,Bi);
}
```

在上述伪代码中，$K$ 是输入密钥；lrot($r_{i-1}$,3)表示将 $r_{i-1}$ 的每个字节循环左移 3 比特；addb($r_i$,$B_i$)表示 $r_i$ 和 $B_i$ 的对应字节进行模 256 加；$B_i$（$i$=2,3,⋯,2$r$+1）是密钥偏移量（Key Biases），每个偏移量由 8 字节组成，记第 $i$ 个偏移量的第 $j$ 字节为 $b[i,j]$，则 $b[i,j]$为

$$b[i,j]=45[45(9i+j) \bmod 257] \bmod 257$$

密钥偏移量的使用减少了弱密钥，也使得层密钥更加随机。

### 1.22.2　SAFER+加密算法

SAFER+加密算法的层函数结构如图 1.22.3 所示。

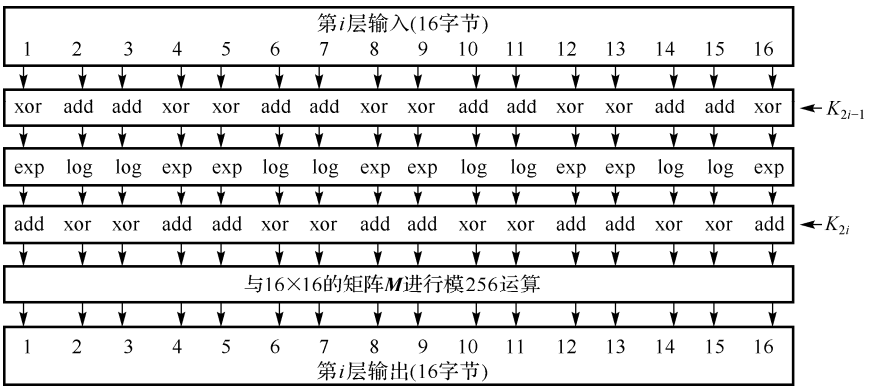

第i层输入(16字节)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
xor add add xor xor add add xor xor add add xor xor add add xor
K2i−1
exp log log exp exp log log exp exp log log exp exp log log exp
add xor xor add add xor xor add add xor xor add add xor xor add
K2i
与16×16的矩阵M进行模256运算
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
第i层输出(16字节)

图 1.22.3　SAFER+加密算法的层函数结构

矩阵 $\boldsymbol{M}$ 如下：

$$\boldsymbol{M}=\begin{bmatrix}
2 & 2 & 1 & 2 & 16 & 8 & 2 & 1 & 4 & 2 & 4 & 2 & 1 & 1 & 4 & 4\\
1 & 1 & 1 & 1 & 8 & 4 & 2 & 1 & 2 & 1 & 4 & 2 & 1 & 1 & 2 & 2\\
1 & 1 & 4 & 4 & 2 & 1 & 4 & 2 & 4 & 2 & 16 & 8 & 2 & 2 & 1 & 1\\
1 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 4 & 2 & 8 & 4 & 1 & 1 & 1 & 1\\
4 & 4 & 2 & 1 & 4 & 2 & 4 & 2 & 16 & 8 & 1 & 1 & 1 & 1 & 2 & 2\\
2 & 2 & 2 & 1 & 2 & 1 & 4 & 2 & 8 & 4 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & 1 & 4 & 2 & 4 & 2 & 16 & 8 & 2 & 1 & 2 & 2 & 4 & 4 & 1 & 1\\
1 & 1 & 2 & 1 & 4 & 2 & 8 & 4 & 2 & 1 & 1 & 1 & 2 & 2 & 1 & 1\\
2 & 1 & 16 & 8 & 1 & 1 & 2 & 2 & 1 & 1 & 4 & 4 & 4 & 2 & 4 & 2\\
2 & 1 & 8 & 4 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 4 & 2 & 2 & 1\\
4 & 2 & 4 & 2 & 4 & 4 & 1 & 1 & 2 & 2 & 1 & 1 & 16 & 8 & 2 & 1\\
2 & 1 & 4 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 8 & 4 & 2 & 1\\
4 & 2 & 2 & 2 & 1 & 1 & 4 & 4 & 1 & 1 & 4 & 2 & 2 & 1 & 16 & 8\\
4 & 2 & 1 & 1 & 1 & 1 & 2 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 8 & 4\\
16 & 8 & 1 & 1 & 2 & 2 & 1 & 1 & 4 & 4 & 2 & 1 & 4 & 2 & 4 & 2\\
8 & 4 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 4 & 2
\end{bmatrix}$$

加密时进行线性变换 $y=x\boldsymbol{M}$，$x$ 为 16 个输入字节，$y$ 为 16 个输出字节。脱密时进行 $x=y\boldsymbol{M}^{-1}$ 变换。

$$\boldsymbol{M}^{-1}=\begin{bmatrix}
2 & -2 & 1 & -2 & 1 & -1 & 4 & -8 & 2 & -4 & 1 & -1 & 1 & -2 & 1 & -1\\
-4 & 4 & -2 & 4 & -2 & 2 & -8 & 16 & -2 & 4 & -1 & 1 & -1 & 2 & -1 & 1\\
1 & -2 & 1 & -1 & 2 & -4 & 1 & -1 & 1 & -1 & 1 & -2 & 2 & -2 & 4 & -8\\
-2 & 4 & -2 & 2 & -2 & 4 & -1 & 1 & -1 & 1 & -1 & 2 & -4 & 4 & -8 & 16\\
1 & -1 & 2 & -4 & 1 & -1 & 1 & -2 & 1 & -2 & 1 & -1 & 4 & -8 & 2 & -2\\
-1 & 1 & -2 & 4 & -1 & 1 & -1 & 2 & -2 & 4 & -2 & 2 & -8 & 16 & -4 & 4\\
2 & -4 & 1 & -1 & 1 & -2 & 1 & -1 & 2 & -2 & 4 & -8 & 1 & -1 & 1 & -1\\
-2 & 4 & -1 & 1 & -1 & 2 & -1 & 1 & -4 & 4 & -8 & 16 & -2 & 2 & -2 & 4\\
1 & -1 & 1 & -2 & 1 & -1 & 2 & -4 & 4 & -8 & 2 & -2 & 1 & -2 & 1 & -1\\
-1 & 1 & -1 & 2 & -1 & 1 & -2 & 4 & -8 & 16 & -4 & 4 & -2 & 4 & -2 & 2\\
1 & -2 & 1 & -1 & 4 & -8 & 2 & -2 & 1 & -1 & 1 & -2 & 1 & -1 & 2 & -4\\
-1 & 2 & -1 & 1 & -8 & 16 & -4 & 4 & -2 & 2 & -2 & 4 & -1 & 1 & -2 & 4\\
4 & -8 & 2 & -2 & 1 & -2 & 1 & -1 & 1 & -2 & 1 & -1 & 2 & -4 & 1 & -1\\
-8 & 16 & -4 & 4 & -2 & 4 & -2 & 2 & -1 & 2 & -1 & 1 & -2 & 4 & -1 & 1\\
1 & -1 & 4 & -8 & 2 & -2 & 1 & -2 & 1 & -1 & 2 & -4 & 1 & -1 & 1 & -2\\
-2 & 2 & -8 & 16 & -4 & 4 & -2 & 4 & -1 & 1 & -2 & 4 & -1 & 1 & -1 & 2
\end{bmatrix}$$

第 $r$ 层的输出结果与层密钥 $K_{2r+1}$ 按字节进行结合，第 1、4、5、8、9、12、13、16 字节进行模 2 加，第 2、3、6、7、10、11、14、15 字节进行模 256 加。

层密钥的生成方法也稍有差异，密钥偏移量由下面两个公式生成：

$$b[i,j]=45[45(17i+j) \bmod 257] \bmod 257,\quad i=1,2,\cdots,17;\ j=1,2,\cdots,16$$

$$b[i,j]=45^{(17i+j)} \bmod 257,\quad i=18,19,\cdots,33;\ j=1,2,\cdots,16$$

密钥生成的伪代码如下：

```
K1=K;
r1=(K1<<8)+sum(K);
for(i=2; i<=2r+1; i++)
{
    ri=lrot(ri-1,3);
    ri=ri<<<8;
    Ki=addb(ri,Bi);
}
```

在上述伪代码中，sum($K$)表示 $K$ 的所有字节的模 2 加之和；addb($r_i$,$B_i$)表示将 $r_i$ 的高 16 字节与 $B_i$ 的对应字节进行模 256 加。

## 1.23　3-Way 加密算法

3-Way 加密算法是由比利时的 J. Daemen、R. Govaexts 和 J. Vandewalle 在 1993 年发表的，其分组长度和密钥长度均为 96 比特，迭代层数为 $n$（可变）。该加密算法并没有对密钥生成层密钥的方法做具体规定，为简单起见，层密钥可由 96 比特密钥与每层的一个不同常数通过模 2 加得到。

$n$ 层迭代变换需要 $n$+1 个层密钥，记为 $K_0,K_1,\cdots,K_n$，加密过程的逻辑框图如图 1.23.1 所示。

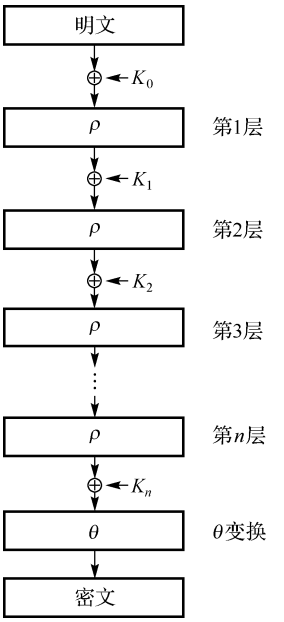



图 1.23.1　加密过程的逻辑框图

1 层至 $n$ 层的 $\rho$ 变换均相同，$\rho$ 由 $\pi_1$、$\pi_2$、$\theta$、$\gamma$ 组成，即

$$\rho=\pi_2\cdot\gamma\cdot\pi_1\cdot\theta$$

表示依次进行 $\theta$、$\pi_1$、$\gamma$、$\pi_2$ 变换或置换。经 $n$ 层变换后，再与第 $n$+1 个层密钥 $K_n$ 进行模 2 加，然后经 $\theta$ 变换后输出密文。

下面介绍基本变换 $\theta$、$\gamma$，以及置换 $\pi_1$、$\pi_2$。注：置换是指将一个分组中的 96 比特的位置进行错乱。

$\theta$、$\gamma$、$\pi_1$、$\pi_2$ 的特性可通过逆序置换 $\mu$ 来刻划，$\mu$ 的作用是将一个分组中的 96 比特逆序排列，即

$$\mu(X_0,X_1,\cdots,X_{94},X_{95})=(X_{95},X_{94},\cdots,X_1,X_0)$$

$\pi_1$、$\pi_2$ 是两个置换，具有下列性质：

$$\pi_1 \cdot \mu \cdot \pi_2=\mu$$

$\gamma$ 是非线性变换，满足：

$$\gamma^{-1}=\mu \cdot \gamma \cdot \mu$$

$\gamma$ 的定义为：

$$\gamma(X_0,X_1,\cdots,X_{95})=(y_0,y_1,\cdots,y_{95})$$

式中，$y_i = \bar{X}_i \oplus \bar{X}_{i+32}X_{i+64}$，$i$=0,1,⋯,95，下角中的“+”号表示模 96 加，$\bar{X}$ 表示 $X$ 取补，$X_{i+32}X_{i+64}$ 表示 $X_{i+32}$ 与 $X_{i+64}$ 相乘，即

$$\gamma(X_0,X_1,\cdots,X_{95}) = (\bar{X}_0 \oplus \bar{X}_{32}X_{64}, \bar{X}_1 \oplus \bar{X}_{33}X_{65},\cdots,\bar{X}_{95} \oplus \bar{X}_{31}X_{63})$$

实际上可看成把 96 比特分成 3 个 32 比特的子分组，$\gamma$ 变换是将这 3 个子分组的对应比特进行逻辑运算，所以这种分组密码的分组长度必须是 3 的整数倍，不同于一般的 64 比特分组。这也许就是 3-Way 名字的由来。

$\gamma$ 变换实现了非线性代替，起到了 DES 加密算法中 $S$ 盒的作用，可看成 32 个相同的 3 比特 $S$ 盒。

$\theta$ 为线性变换，满足：

$$\theta^{-1}=\mu \cdot \theta \cdot \mu$$

$\theta$ 的定义为：

$$\theta(X_0,X_1,\cdots,X_{95})=(y_0,y_1,\cdots,y_{95})$$

式中，$y_i=X_i \oplus X_{i+8} \oplus X_{i+16} \oplus X_{i+24} \oplus X_{i+40} \oplus X_{i+48} \oplus X_{i+80}$，下角中的“+”号表示模 96 加。$\theta$ 是可用多项式乘法来刻划的线性映射。将 $(X_0,X_1,\cdots,X_{95})$ 看成多项式 $f(z)=\sum_{i=0}^{95} X_i z^i$，令 $e(z)=1+z+z^2+z^3+z^5+z^6+z^{10}$，计算多项式乘积 $g(z)=f(z)e(z^8) \mod z^{96}=\sum_{i=0}^{95} g_i z_i$，则 $g(z)$的系数就是$(X_0,X_1,\cdots,X_{95})$经线性变换 $\theta$ 后的函数值。

## 1.24 SHARK 加密算法

SHARK 加密算法是 1996 年由比利时的五位学者提出的一种非 Feistel 结构的分组密码，其分组长度为 64 比特，密钥不超过 128 比特，迭代层数为 8。和 DES 加密算法相似之处是，SHARK 加密算法也有 8 个 $S$ 盒，每层的变换也是由加乱、$S$ 盒和扩散构成的。和 DES 加密算法的不同之处是，SHARK 加密算法并不把输入分组分成左右两部分，而是每层对整个输

入分组进行变换；扩散不是通过移位来实现的，而是通过一个最大距离可分纠误编码实现的；层密钥的生成由自身加密来完成。SHARK 加密算法便于软件实现。

SHARK 加密算法的结构如图 1.24.1 所示。

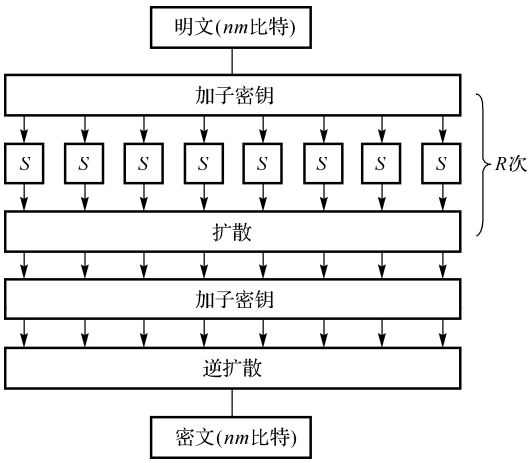



图 1.24.1　SHARK 加密算法的结构

明文分组经过 $R$ 次同样的层变换后，与子密钥进行一次结合，然后进行一次逆扩散变换，即可输出密文。层变换由三个独立的部分组成，第一部分是输入与层密钥的结合过程；第二部分是将输入分成 8 个 8 比特的字，分别经过 8 个相同 $S$ 盒的非线性代替，输出 8×8 比特的字；第三部分是一个线性扩散层。这三个部分是互相独立的。

线性扩散层采用了如下定义的线性变换 $\gamma$：

$$\gamma：\mathrm{GF}(2^m)^n \rightarrow \mathrm{GF}(2^m)^n：X \rightarrow Y=\boldsymbol{B}\cdot X$$

式中，$n$ 为 $S$ 盒的个数，$m$ 为每个 $S$ 盒的输入比特，在 SHARK 加密算法中，$n$ 和 $m$ 都为 8。$\boldsymbol{B}$ 为一个 $n$ 阶方阵，$[\boldsymbol{I}_{n\times n}\boldsymbol{B}_{n\times n}]$是 Galois 域 $\mathrm{GF}(2^m)$上的最大距离可分纠误编码$(2n, n, n+1)$的生成矩阵的标准形式。

上面定义的线性变换 $\gamma$ 为雪崩效应最好的线性变换。

非线性扩散层由 8 个相同的 $S$ 盒组成，每个 $S$ 盒都是 $\mathrm{GF}(2^m)$上的非线性代替 $s$：$Y=X^{-1}$。在它的差分统计表 $E_{ij}=\#\{x|s(x)\oplus s(i\oplus x)=j\}$内，每一行都只有一个最大值 4。为了消除固定点 0→0、1→1，在输出时又加了一个可逆仿射变换。

为了实施方便，将层变换的全部内容合并为 $n$ 次查表。

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \cdots \\ Y_n \end{bmatrix} = T_1[X_1]+T_2[X_2]+\cdots+T_n[X_n]$$

$$T_i[X_i]=\begin{bmatrix} b_{1i}\cdot s_i[X_i] \\ b_{1i}\cdot s_i[X_i] \\ \cdots \\ b_{1i}\cdot s_i[X_i] \end{bmatrix}$$

式中，$X_i$ 为第 $i$ 个 $S$ 盒的输入，“·”和“+”为 GF($2^m$)上的乘法和加法。

SHARK 加密算法共需 $R$+1 个 $mn$ 比特的子密钥，记为 $K_i$, $i$=1,2,⋯,$R$+1。子密钥与层输入的结合方式可以有两种：一种是模 2 加；另一种是用子密钥对输入进行一次仿射变换，即

$$Y=\boldsymbol{k}_i\cdot X\oplus K_i$$

式中，$X$ 为 $mn$ 比特输入，$\boldsymbol{k}_i$ 为一个与密钥有关的 GF($2^m$)上的 $n\times n$ 可逆矩阵，它可以提高层密钥的熵，若为了节省运算量，可用对角矩阵。

子密钥生成方法如下：

（1）对子密钥进行初始化。用代替表 $T_0$ 的前 $R$+1 个 $nm$ 比特作为 $K_i$ 的初态。

$$\boldsymbol{k}_i=\boldsymbol{I}_{n\times n}$$

（2）将密钥反复延长，得到一个 2($R$+1)$mn$ 比特的长串，作为 SHARK 加密算法的输入。

（3）用 SHARK 加密算法以 64 比特 CFB 模式进行加密，在 2($R$+1)$mn$ 输出比特中，前（$R$+1）个 $mn$ 比特作为 $R$+1 个 $K_i$ 值，后面（$R$+1）个 $mn$ 比特记为($R$+1)$n$ 个 GF($2^m$)上的值，按顺序作为（$R$+1）个矩阵 $k_i$ 的对角元素。如果某个值为零，就丢弃该值，后面的值自动前移 1 位，最后添加上对全零串加密得到的值。

SHARK 加密算法的脱密过程同加密过程完全一致，但需要改变查表内容。

## 1.25 SQUARE 加密算法

SQUARE 加密算法是由比利时的 Joan Daeman、LarsKundsen 和 Vincent Rijmen 于 1997 年提出的，其分组长度和密钥长度为 128 比特，迭代层数为 8 层。SQUARE 加密算法不属于 Feistel 结构，它的结构类似于 3-Way 加密算法和 SHARK 加密算法的结构。

将 128 比特的输入 $a$ 分成 16 个字节，将这 16 个字节排成 4×4 的方阵，记成[$a_{ij}$]，0≤$i$、$j$≤3，SQUARE 加密算法的层函数对整个方阵以单个比特、字节行或列为单位进行加乱、转置、代替和多项式乘法 4 次变换，这也就是 SQUARE 名字的由来。

SQUARE 加密算法的结构框图如图 1.25.1 所示。

层函数 $\rho=\sigma\cdot\pi\cdot\gamma\cdot\theta$，其中 $\sigma$、$\pi$、$\gamma$、$\theta$ 为 4 个不同的变换。

$\theta$ 是一个线性变换，它对输入方阵中的每行进行独立的操作，即

$$\theta：b=\theta(a) \Leftrightarrow b_{ij}=c_j a_{i0} \oplus c_{j-1} a_{i1} \oplus c_{j-2} a_{i2} \oplus c_{j-3} a_{i3}，\quad 0 \leqslant i,j \leqslant 3$$

式中，乘法是有限域 GF($2^8$)中的乘法，$c$ 的下角模 4 运算。

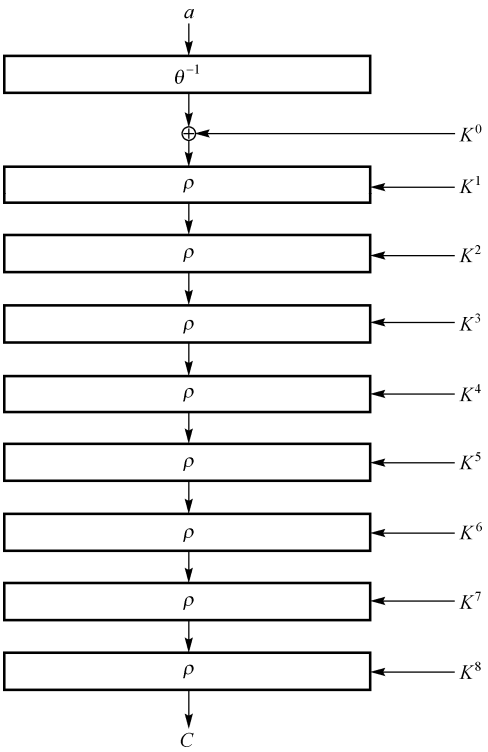



图 1.25.1　SQUARE 加密算法的结构框图

为了直观起见，可以用多项式乘法来表示 $\theta$ 变换：

$$\theta：b=\theta(a) \Leftrightarrow b_i(x)=c(x)a_i(x) \bmod (x^4 \oplus 1)，\quad 0 \leqslant i \leqslant 3$$

式中，$a_i(x)=a_{i0} \oplus a_{i1}x \oplus a_{i2}x^2 \oplus a_{i3}x^3$，$0 \leqslant i \leqslant 3$。

$$c(x)=c_j \oplus x^j$$

$c(x)$的取法是 $\theta$ 变换的核心，$\theta$ 变换的差分特征概率和线性特征概率均取决于 $c(x)$，为使两个概率比较低，$c(x)$必须有很高的扩散能力，即 $c(x)$的分支数（Branch Number）要很大。$c(x)$的构造采用了最大距离可分（Maximum Distance Separable Code, MDS-Code）码，由于 $c(x)$的系数矩阵是 4×4 的循环矩阵，因此 $c(x)$的取值自由度比较大，取：

$$c(x)=2_X \oplus 1_X \cdot x \oplus 1_X \cdot x^2 \oplus 3_X \cdot x^3$$

$\theta^{-1}$ 是 $\theta$ 的逆变换，$\theta^{-1}$ 的系数多项式 $d(x)$由 $c(x)$唯一确定，即

$$d(x)=E_X \oplus 9_X \cdot x \oplus D_X \cdot x^2 \oplus B_X \cdot x^3$$

$\gamma$ 是一个可逆的非线性变换，它以字节为单位进行运算，其直观表现形式是一个 8 比特的可逆代替表 $S_\gamma$，即

$$\gamma:\quad b=\gamma(a)\Leftrightarrow b_{ij}=S_\gamma(a_{ij}),\quad 0\leqslant i、j\leqslant 3$$

$S_\gamma$ 的选取也是为了有效地抵制差分分析和线性分析，它采用了有限域 GF($2^8$)中的映射 $x\rightarrow x^{-1}$，该映射具有极好的特性，即最佳的选择。但由于该映射的描述太简单，而层函数的其他变换过程的描述也很简单，这样容易用代数的手段分析，因此对映射 $x\rightarrow x^{-1}$ 的输出又进行了复杂的仿射变换。

$\pi$ 变换是对字节矩阵$[a_{ij}]$进行行列转置，即

$$\pi:\quad b=\pi(a)\Leftrightarrow b_{ij}=a_{ji},\quad 0\leqslant i、j\leqslant 3$$

$$\pi^{-1}=\pi$$

$\sigma$ 变换是模 2 加层密钥 $K^i$ 的过程，即

$$\sigma[K^i]:\ b=\sigma[K^i](a)\Leftrightarrow b=a\oplus K^i$$

$$\sigma^{-1}[K^i]=\sigma[K^i]$$

SQUARE 加密算法可用下面的公式来表示，即

$$\mathrm{SQUARE}(K)=\rho[K^8]\rho[K^7]\rho[K^6]\rho[K^5]\rho[K^4]\rho[K^3]\rho[K^2]\rho[K^1]\sigma[K^0]\theta^{-1} \qquad (1\text{-}25\text{-}1)$$

SQUARE 加密算法的层密钥由输入密钥按迭代规律生成，即

$$K^{i+1}=\psi(K^i),\qquad i=0,1,\cdots,8$$

可将 128 比特的层密钥看成 4 个 32 比特的字 $K_0^i$、$K_1^i$、$K_2^i$、$K_3^i$。迭代规律如下:

$$K^0=K$$

$$K_0^{i+1}=K_0^i\oplus \mathrm{rotl}(K_3^i)\oplus C_i$$

$$K_1^{i+1}=K_1^i\oplus K_0^{i+1}$$

$$K_2^{i+1}=K_2^i\oplus K_1^{i+1}$$

$$K_3^{i+1}=K_3^i\oplus K_2^{i+1}$$

式中，rotl($a$)表示将 $a$ 循环左移 1 字节；$C_i$ 是层常数，也是由迭代生成的，$C_0=1_X$，$C_i=2_X\cdot C_{i-1}$，$i=1,2,\cdots,8$。

生成层密钥所用的 $\psi$ 函数具有极强的扩散性，因此能够有效地抵御相关密钥攻击和其他攻击。层常数的使用在一定程度上减弱了层变换之间和层之间的对称性。

脱密过程所用的层密钥生成过程如下：

$$K^0=K$$

$$K_3^{i+1} = K_3^i \oplus K_2^i$$

$$K_2^{i+1} = K_2^i \oplus K_1^i$$

$$K_1^{i+1} = K_1^i \oplus K_0^i$$

$$K_0^{i+1} = K_0^i \oplus \text{rotr}(K_3^i) \oplus C_i$$

式中，rotr($a$)表示将 $a$ 循环右移 1 字节。

SQUARE 加密算法的脱密过程同加密过程几乎完全一致。由式（1-25-1）可以很容易地推算出下式：

$$\text{SQUARE}^{-1}(K)=\rho[K^8]\rho[K^7]\rho[K^6]\rho[K^5]\rho[K^4]\rho[K^3]\rho[K^2]\rho[K^1]\sigma[K^0]\theta$$

式中，

$$K^i=\theta(K^{8-i})$$

$$\rho[K^i]=\sigma[k^i]\pi\gamma^{-1}\theta^{-1}$$

## 1.26　MMB 加密算法

MMB 加密算法是由比利时的 Joan Daeman 提出的，它基于模乘法运算的分组密码，其分组长度和密钥长度均为 128 比特，迭代层数为 6。整个加密算法以 32 比特为单位进行运算，因此非常适合在 32 位的计算机上运行。

MMB 加密算法的层变换为：

$$X_{i+1}=F(X_i \oplus K_i), \qquad i=0,1,2,3,4,5,6$$

6 个层密钥都为 128 比特，层密钥的生成非常简单：

$$K_1=K_4=K$$

$$K_2=K_5=K<<<32$$

$$K_3=K_6=K<<<64$$

式中，符号“<<<”表示循环左移位。

层函数 $F$ 由模 2 加和模 $2^{32}-1$ 乘法两种运算组成，$X_i \oplus K_i$ 作为第 $i$ 层的层函数 $F$ 的输入。将层函数 $F$ 的 128 比特输入从左到右分成 4 个 32 比特的字，记为 $x_0$、$x_1$、$x_2$、$x_3$，伪代码如下：

```
for(i=0; i<4; i++)
    x_i=C_i·x_i mod (2^32-1)
    如果(x_0&1)=1, x_0=x_0⊕C
    如果(x_3&1)=0, x_3=x_3⊕C
for(i=0; i<4; i++)
    x_i=x_{i-1}⊕x_i⊕X_{i+1}
```

上述伪代码中的下角均为模 4 运算。$C$ 和 $C_i$（$0 \leqslant i \leqslant 3$）的取值为：

$$C=0x2aaaaaaa$$

$$C_0=0x025f1cdb$$

$$C_1=2C_0$$

$$C_2=2^3C_0$$

$$C_3=2^7C_0$$

MMB 加密算法的脱密过程同加密过程一致，只是将层函数 $F$ 中 $C_i$ 替换成 $C_i^{-1}$ 即可。

## 1.27 XMX 加密算法

XMX 加密算法是 1997 年由法国的 David 等四人提出的一种对称的非 Feistel 结构的分组密码，它没有 DES 加密算法所具有的 $S$ 盒和 $P$ 置换表，也不需要生成子密钥，只使用了模乘法和模 2 加操作，对整个输入进行多次迭代，全部加密过程可以用一个简洁的公式来表示。XMX 加密算法的设计充分利用了硬件和软件的优点，因此非常适合在公钥库和带有算术协处理器的微控制器等固件设备上使用。

算法描述如下：

XMX 加密算法的分组长度 $L$ 为 512 比特，迭代层数 $r$ 为 8，密钥 $K$ 是一个长度为 $2r+1$ 的数组，数组元素为 $L$ 比特长的数，即密钥 $K$ 的长度为 $(2r+1)L$ 比特，即

$$K=\{a_1,b_1,\cdots,a_r,b_r,a_{r+1}\}$$

层函数 $F$ 的定义为：

$$F(a,b)(m)=(m\odot a)\cdot b \bmod n$$

式中，

$$x\odot y=\begin{cases} x\oplus y, & x\oplus y<n \\ x, & \text{其他} \end{cases}$$

模数 $n$ 是一个小于 $2^L$ 的奇数，$a$、$b$、$m$ 均小于 $n$，$(b,n)=1$。

XMX 加密算法的加密过程可以用下面的公式来表示，即

$$\mathrm{XMX}(K,m)=(F(a_r,b_r)(F(a_{r-1},b_{r-1})(\cdots(F(a_1,b_1)(m))\cdots)))\odot a_{r+1}$$

XMX 加密算法的脱密过程如下：从 $F$ 函数的定义可以得到 $F^{-1}$，即

$$F^{-1}(a,b)(m)=(m\cdot b^{-1} \bmod n)\odot a$$

因此

$$\begin{aligned}\mathrm{XMX}^{-1}(K,c)&=(F^{-1}(a_1,b_1)(F^{-1}(a_2,b_2)(\cdots(F^{-1}(a_r,b_r)(c\odot a_{r+1}))\cdots)))\\&=(F(a_2,b_1^{-1}\bmod n)(F(a_3,b_2^{-1}\bmod n)(\cdots(F(a_{r+1},b_r^{-1}\bmod n)(c))\cdots)))\odot a_1\end{aligned}$$

即

$$\mathrm{XMX}^{-1}(K,x)=\mathrm{XMX}(K^{-1},x)$$

式中，

$$K^{-1}=\{a_{r+1},b_r^{-1}\bmod n,a_r,\cdots,b_1^{-1}\bmod n,a_1\}$$

从上面可以看出，XMX 加密算法的脱密过程同加密过程是完全一致的。

在具体的 XMX 加密算法中，取

$$k=\{S,S,S,S,S,S,S,S,S\oplus S^{-1},S,S^{-1},S,S^{-1},S,S^{-1},S,S^{-1}\}$$

这样便于存储，且

$$k^{-1}(S)=k(S^{-1})$$

要求 $S$ 满足下列条件：

（1）$S\in Z_n^*$，且 $L/2-\log_2 L<W(S)<L/2+\log_2 L$，即 $S$ 的 Hamming 重量非常接近 $L/2$。

（2）$(S,n)=1$，且 $S>2^{L-2}$。

模数 $n$ 的选取限制条件比较少，David 等人选取 $n=2^{512}-1$；但在实际应用中可以考虑下面两个因素：

（1）如果 $n$ 取素数，密钥的选取将大大简化。

（2）如果 $n$ 取 RSA 模，即 $n$ 等于两个大素数的乘积，将大大方便内存管理。

## 1.28　Madryga 加密算法

Madryga 加密算法是加拿大帝国商业银行的 W. E. Madryga 在 1984 年 IFIP（International Federation for Information Processing）国际计算机安全会议上提出的，它是为加拿大帝国商业银行作为替代 DES 的分组密码而设计的。

Madryga 加密算法不同于一般的分组密码，它没有明确的分组概念，在加/脱密时，对整个消息从头至尾进行多次变换，每次基本变换是对 3 字节进行移位和模 2 加运算，每次基本变换后前进 1 字节，再做下一次基本变换，所以 Madryga 加密算法可看成以字节为单位进行运算，并在相邻字节间有连接关系。我们可以把一份明文消息看成一个分组，从这个意义上讲，Madryga 加密算法的分组长度就是明文的长度。设明文消息的字节数为 $L_t$，则 $L_t$ 是大于 3 的任意整数，所以 Madryga 加密算法不需要对明文进行填补。Madryga 加密算法的密钥长度是可变的，一般采用 8 字节，我们把密钥的字节数记为 $L_k$，除密钥外还有 $L_k$ 字节的常数密钥用于避免弱密钥。

Madryga 加密算法的优点是对于软件、微码（Microcode）以及硬件等各种实现手段，都能做到速度快，且价格便宜，而 DES 加密算法只适合于硬件实现。此外，由于密钥长度和明文长度的可变性，使 Madryga 加密算法有很大的灵活性，可适应各种不同的需求。该算法的强度依赖于参数的选择，计算上的强度等价于 DES 加密算法。

Madryga 加密算法的基本变换只有模 2 加和循环移位两种，在密钥的作用下反复进行模 2 加和移位来实现密码变换。整个算法由两个嵌套的循环构成，内循环重复 $L_t$ 次，外循环是对内循环重复 8 次（或更多次），每次称为一轮操作，每轮操作都对整个消息做一遍内循环。若把整个明文消息看成一个明文分组，则一轮操作就类似于迭代分组密码中的一个层变换。

内循环重复 $L_t$ 次，每次重复都在一个包含 3 字节的工作窗口（也称为工作框架）中进行运算。把明文消息看成一个头尾相连的轮，最后一个字节后面紧接着第一个字节，每次运算后工作窗口在明文消息上向前移动 1 字节，循环 1 周，共重复 $L_t$ 次运算，完成一次内循环，即一轮操作。

工作窗口中共有 3 字节，自左至右（由高位到低位）分别记为 $W_1$、$W_2$、$W_3$，3 字节分成两部分：高位 2 字节和低位 1 字节。高位 2 字节进行错位（Transposition）运算，由循环移位来实现，移位数可变，由输入数据和密钥决定；低位 1 字节进行变换（Translation）运算，通过与相应的密钥进行模 2 加来实现，低位 1 字节可还用来决定循环移位数。

Madryga 加密算法的逻辑框图如图 1.28.1 所示。

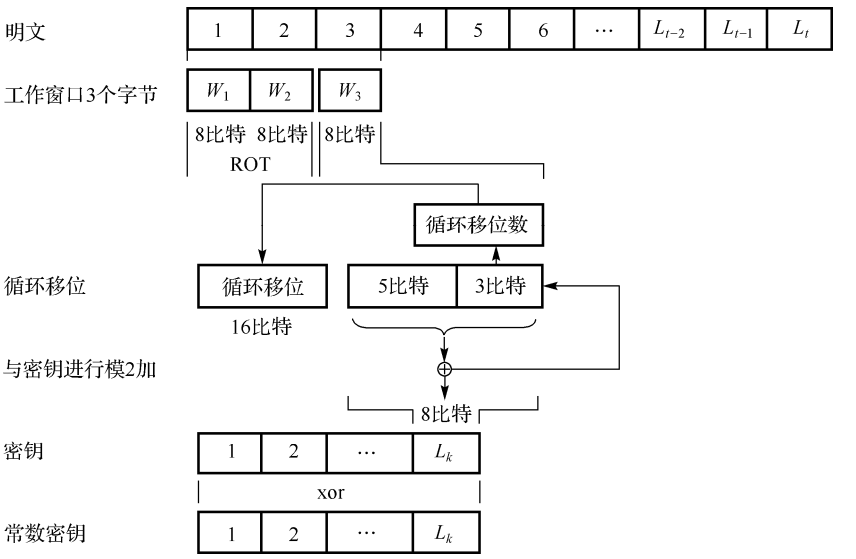



图 1.28.1　Madryga 加密算法的逻辑框图

Madryga 加密算法的加密过程如下：进行 $n$ 轮（$n \geqslant 8$）操作，每轮操作的工作窗口从明

文倒数第 2 字节开始，每次循环右移 1 字节，循环一周，到最后第 3 字节结束，对工作窗口中的三个字节 $W_1$、$W_2$、$W_3$ 进行下述变换：

① 密钥循环右移 3 比特，把常数密钥的对应位模 2 加到密钥上。

② 保留 $W_3$ 的低 3 比特，并作为高位字节的循环移位数。

③ 密钥的低位字节与 $W_3$ 进行模 2 加，改变窗口中的 $W_3$，实现变换，并隐蔽循环移位数。

④ 根据循环移位数，把窗口中的 $W_1$ 和 $W_2$ 连接起来循环左移，实现错位。

在脱密时，先由给定的密钥和常数密钥计算出加密时密钥的最后状态，然后进行 $n$ 轮（$n \geqslant 8$）操作，每一轮操作的工作窗口从密文的最后 3 字节开始，每次循环左移 1 字节，循环一周。对工作窗口中的 3 字节进行下述变换：

① 密钥的最低位字节与 $W_3$ 进行模 2 加，恢复出循环移位数和先前的数据。

② $W_3$ 的低 3 比特进行循环移位数，把 $W_1$ 和 $W_2$ 连起来循环右移。

③ 密钥与常数密钥进行模 2 加，改变密钥状态，然后将密钥循环左移 3 比特。

Madryga 加密算法的基本变换是由与密钥进行模 2 加，以及在密钥和数据控制下的错乱构成的，因为工作窗口每次只前进 1 字节，所以会覆盖前面的模 2 加和错乱，经 $n$ 轮操作后，可使每个比特的错乱范围向左可达 $n$ 字节、向右可达 $7n$ 字节，并使密钥在数据和密钥的影响下反复与数据累加。

Madryga 加密算法中的常数密钥为“0”“1”平衡的随机序列，用于消除弱密钥。操作的轮数至少为 $\lceil (L_k \times 8) / (3 \times \min(L_t)) + 0.5 \rceil$，以保证每个密钥位都参与加密过程，并使所有数据以很高的概率用于移位数。建议密钥采用 8 字节，操作采用 $(8 \times 8) / (3 \times 3) + 0.5 = 8$ 轮。一轮操作的密度是很低的，当密钥长度和操作轮数增加时，密度会增强，应该根据数据的具体情况来调整轮数和密钥长度，以获得理想的安全性。

## 1.29　FBCP 加密算法

FBCP（Fast Block Cipher Proposal）加密算法是由 RSA 实验室的 Kaliski 和 Robshaw 于 1993 年设计的，它是一种新的分组加密算法，不是 Feistel 结构的。该加密算法的分组长度为 256×32 比特，密钥长度为 80 比特，迭代层数为 4。FBCP 加密算法有两个显著特点：一是分组长度很大；二是层函数的设计采用了著名的 Hash 函数 MD5 的思想。

FBCP 加密算法。将 256×32 比特的明文分组按顺序分成 256 个 32 比特的字并放在数组 $m$[256]中。下面是 FBCP 加密算法的具体加密过程：

① 对数组 $m$[256]进行一次 $P$ 置换，变成新的 $m$[256]数组。$P$ 置换由层密钥生成，生成方法将在后面介绍。

② 进行如下述伪代码所示的 4 层迭代过程:

```
for(r=0; r<4; r++)
for(g=0; g<64; g++)
{
    k=4*g;
    A=m[k<<<2r];
    B=m[(k+1)<<<2r];
    C=m[(k+2)<<<2r];
    D=m[(k+3)<<<2r];
    for(s=0; s<8; s++)
    {
        A=(A⊕ F_r(B,C,D,X_{512r+8g+s}))<<<5;
        D=C;C=B;B=A;A=D;
    }
    m[k<<<2r]=A;
    m[(k+1)<<<2r]=B;
    m[(k+2)<<<2r]=C;
    m[(k+3)<<<2r]=D;
}
```

迭代结束后，数组 $m$[256]中所保存的就是密文。$X_i$（$0 \leqslant i \leqslant 2047$）是 2048 个 32 比特的字，它和 $P$ 置换一样都是由密钥生成的，由 80 比特密钥生成 $P$ 置换和 $X_i$（$0 \leqslant i \leqslant 2047$）的方法如下：

① 将 80 比特的密钥 $K$ 按字节放在数组 $K$[10]中，然后利用下面的递归公式把该密钥扩展成 256 字节，放在数组 $K$[256]中。

$$K[a+10]=K[a+8] \oplus K[a+4] \oplus K[a+3] \oplus K[a], \qquad 0 \leqslant a \leqslant 245$$

② 令 $P$ 置换为单位置换。

③ 重复下面伪代码所示的过程两次，即生成 $P$ 置换。

```
for(i=256; i>0; i--)
{
    m=(K[256-i]+K[257-i]) mod i;
    K[257-i]=K[257-i]<<<3;
    交换 P[m]和 P[i-1];
}
```

对密钥数组 $K$[10]进行变换，使得 $K[i]=k[9-i]$，$i$=0,1,2,…,9，然后按生成 $P$ 置换的方法生成 $T$ 置换，利用 $T$ 置换来合成 $X$，合成方法如下：

当 $0 \leqslant i \leqslant 255$ 时，

$$X_i=T_i \parallel T_{i+1} \parallel T_{i+2} \parallel T_{i+3}$$

当 $256 \leqslant i \leqslant 511$ 时，

$$X_i=T_{i-256} \parallel T_{i-256+2} \parallel T_{i-256+4} \parallel T_{i-256+6}$$

当 512≤$i$≤1023 时，

$$X_i=X_{i-512}<<<3$$

当 1024≤$i$≤1535 时，

$$X_i=X_{i-1024}<<<7$$

当 1536≤$i$≤2047 时，

$$X_i=X_{i-1536}<<<19$$

层函数 $F_r$ 的构造采用了 Hash 函数 MD5 的思想，即

$$F_r(B,C,D,X_i) = [B+f_r(B,C,D)+X_i] \bmod 2^{32}$$

式中，$f_r$ 的四种形式如下：

$$f_0(B,C,D)=(B\&C)|(\sim B\&D)$$
$$f_1(B,C,D)=(B\&D)|(C\&\sim D)$$
$$f_2(B,C,D)=B\oplus C\oplus D$$
$$f_3(B,C,D)=C\oplus (B\&\sim D)$$

上述算法下角中的运算均为模 256 运算。

FBCP 加密算法的脱密过程同加密过程完全一样。

## 1.30　VINO 加密算法

VINO 加密算法是由意大利的 Adina Di Porto 和 William Wolfowicz 于 1993 年 12 月在剑桥算法研讨会上发表的，是一种具有可变置换的分组密码。该密码体制采用了维诺格拉多夫（Vinogradov）提出的可变置换，并用 VINO 来命名。

VINO 加密算法的输入、输出分组均为 64 比特，密钥长度为 128 比特。

### 1．$V$ 置换（可变置换）

设序列 $b=(b_0,b_1,b_2\cdots)$，$s$ 为一正整数，称 $b^{(s)}=(b_0,b_s,b_{2s}\cdots)$为序列 $b$ 的 $s$ 采样。

设 $n$、$m$ 为整数，考虑序列

$$a=(a_0,a_1,a_2,\cdots,a_{n-1})$$

将 $a$ 按正序 $a_0,a_1,a_2,\cdots,a_{n-1}$ 连接反序 $a_{n-1},a_{n-2},\cdots,a_1$，再正序，再反序，…，不断重复，即可得到序列

$$b=(b_0,b_1,b_2\cdots)=(a_0,a_1,a_2,\cdots,a_{n-1},a_{n-2},\cdots,a_1,a_0,a_1\cdots)$$

对 $b$ 进行 $m$ 采样，可得序列

$$a' = b^{(m)} = (b_0, b_m, b_{2m}, b_{3m}\cdots)$$

直至得到 $n$ 个元素。显然，当（$m$,2$n$−1）=1 时，$a'$中的元素是 $a$ 的元素的置换，称对 $a$ 施以 $V_{(m)}$置换得 $a'$，记为：

$$V_{(m)}(a)=a'$$

Vinogradov 在 1945 年编写的 *Elements of number theory*（数论的元素）一书中定理：

$n$、$m$ 为正整数，（$m$, 2$n$−1）=1，序列 $a$ 如上所设，对 $a$ 施以 $V_{(m)}$置换可得 $a'$，对 $a'$施以 $V_{(m)}$置换得 $a''$，…，如此下去，则经 $k$ 次 $V_{(m)}$置换后可得原序列 $a$，当且仅当

$$m^k \equiv \pm 1 (\text{mod } 2n-1)$$

**证明**：由 $a$ 生成的序列 $b$ 的周期为 2$n$−1，定义

$$V_{(m)}^{(i)}(a) = V(m)[V(m)^{i-1}(a)]$$

由采样的定义可知：

$$V_{(m)}^{(i)}(a) = V_{(m')}(a)$$

式中，$m'$是 $m^i$ mod 2$n$−1 剩余，对 $a$ 施以 $V_{(1)}$置换可得原序列 $a$，上述定理可由数论结果求得。

例如，$n$=6，$m$=3，(3,11)=1，$a$=(1,2,3,4,5,6)，用 Vinogradov 方法构造 1、2、3、4、5、6 的置换。

序列

$$b=(1,2,3,4,5,6,6,5,4,3,2,1,2,3,4,5,6,6,5,4,3,2\cdots)$$

$$a'=V_{(m)}(a)=b^{(m)}=(1,4,6,3,2,5)，\ m=3$$

由 $3^{10} \equiv 1$ (mod 11)，其中 11=2$n$−1，可计算得 $3^5 \equiv 1$ (mod 11)。由上述定理对 $a$ 进行 5 次 $V$ 置换仍可得 $a$，即

$$V_{(3)}^5(a) = V_{(m)}(a) = a$$

可知 $V_{(3)}^5 = I$，此处 $I$ 为单位置换。

例如：

$a$ = (1,2,3,4,5,6)

$a'$ = $V_{(3)}(a)$=(1,4,6,3,2,5)

$b'$ = (1,4,6,3,2,5,5,2,3,6,4,1,4,6,3,2,5…)

$a''$ = $V_{(3)}(a')$=(1,3,5,6,4,2)

$b''$ = (1,3,5,6,4,2,2,4,6,5,3,1,3,5,6,4,2…)

$a'''$ = $V_{(3)}(a'')$=(1,6,2,5,3,4)

$b'''$ = (1,6,2,5,3,4,4,3,5,2,6,1,6,2,5,3,4…)

$$a'''' = V_{(3)}(a''')=(1,5,4,2,6,3)$$
$$b'''' = (1,5,4,2,6,3,3,6,2,4,5,1,5,4,2,6,3\cdots)$$
$$a''''' = V_{(3)}(a'''')=(1,2,3,4,5,6)$$

可见

$$V(a)=a$$

$V_{(3)}^{5}(a)$ 的逆置换是 $V_{(3)}^{4}(a)$。

### 2．VV 置换

下面讨论 $V$ 置换作用在（0,1）序列的情形。设（0,1）序列 $I$ 经 $V_{(m)}$置换得 $O$ 序列，$I$、$O$ 序列均为 16 比特，$m$ 由 $H$ 的子序列 $H_i$ 提供，$H$ 也是 16 比特序列，即

$$H = (h_0,h_1,h_2,h_3,|\ h_4,\cdots,h_7,|\ h_8,\cdots,h_{11}\ |\ h_{12},\cdots,h_{15})$$
$$= (H_0,H_1,H_2,H_3)$$

式中，$H_i$ 是由 4 比特（$h_{4i},h_{4i+1},h_{4i+2},h_{4i+3}$）决定的二进制数，即

$$0\leqslant h_i\leqslant 15,\qquad i=0,1,2,3$$

以 $H$ 为参数对输入 $I$ 进行下面四步工作：

（1）对 $I$ 施以 $V_{(H_0)}$置换得 $I'$。

（2）对 $I'$循环左移 $H_1$ 比特可得 $I''$。

（3）对 $I''$施以 $V_{(H_2)}$置换得 $I'''$。

（4）将 $I'''$循环左移 $H_3$ 比特可得输出序列 $O$。

这四步合起来称为以 $H$ 为控制参数对 $I$ 进行 VV 置换得输出 $O$，记为 $VV_H(I)=O$，如图 1.30.1 所示。

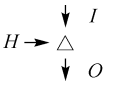



图 1.30.1　VINO 加密算法 VV 置换（一）

### 3．VINO 加密算法的层变换

VINO 加密算法的层变换如图 1.30.2 所示，图中，⊕ 表示对应比特模 2 加，⊞表示 mod $2^n$ 整数加。第 $r$ 层输入为 $X_{1r},X_{2r},X_{3r},X_{4r}$，其中 $X_{ir}$ 都是 16 比特的子分组；第 $r$ 层输出为 $Y_{1r},Y_{2r},Y_{3r},Y_{4r}$，其中 $Y_{ir}$ 都是 16 比特的子分组；第 $r$ 层输入为 $K_{ir}(i=1,2,\cdots,8)$，$K_{ir}$ 都是 16 比特的子分组。

每层的输入 $M_r$ 首先经以 $K_r$ 的前 64 比特为控制参数的 VV 置换（如图 1.30.3 所示），可得 $M_r'=(X_{1r},X_{2r},X_{3r},X_{4r})$。

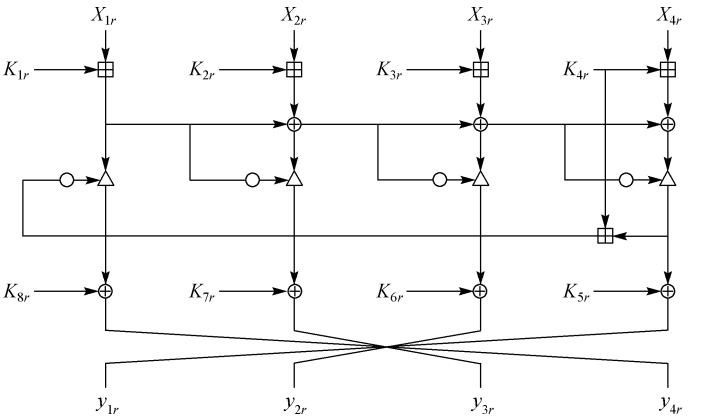



图 1.30.2　VINO 加密算法的层变换

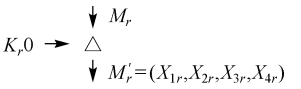



图 1.30.3　VINO 加密算法 VV 置换（二）

在整个加密过程中，消息 $M$ 须经 $t$ 次如图 1.30.1 所示的层变换，得到 $M'$，最后对 $M'$进行一次 VV 置换，输出密文分组 $C$。考虑加密速度及安全，取 $t$=4 较为合适。

### 4．VINO 算法层密钥的产生

密钥 $K$ 的长度为 128 比特，第 $r$ 层的层密钥记为 $K_r$，第 1 层的层密钥 $K_1$=$K$，第 $j$ 层的层密钥 $K_j$（$j$=2,3…）按下列步骤产生：

设 $K_{j-1}=(K_{j-1}^1,K_{j-1}^2,\cdots,K_{j-1}^8)$，每个 $K_{j-1}^i$ 为 16 比特。

（1）以 $K_{j-1}^i$ 为控制参数对 $K_{j-1}^i$ 自身进行 VV 置换，产生 $K_{j-1}^{i'}$，$i$=1,…,8。令 $K'_{j-1}=(K_{j-1}^{1'},K_{j-1}^{2'},\cdots,K_{j-1}^{8'})$。

（2）将 $K_{j-1}$ 左移 1 比特，右端最后一位补 0，第 0,2,4…比特用其补码代替，可得 $K''_{j-1}$。

（3）第 $j$ 层的层密钥 $K_j=K'_{j-1}\oplus K''_{j-1}$。

这种新的加密体制对于密钥、明文的选择具有高度的自由，不存在弱密钥。

VINO 加密算法的初步分析：

（1）VINO 加密算法采用了 Vinogradov 提出的可变置换，增加了密码体制内部的可变因素，这对于固定用同一置换来说不失为一种改进，会给差分攻击方法增加难度。另外，也不难看出，Vinogradov 提出的可变置换的变化是有限的，至多只有 $n$–1 个互异的置换，且由一个置换的方幂生成。如上面的例中，只有以下 5 个不同的置换：

$$a=V_{(1)}(a)=(1,2,3,4,5,6)$$

$$V_{(2)}(a)=(1,3,5,6,4,2)$$
$$V_{(3)}(a)=(1,4,6,3,2,5)$$
$$V_{(4)}(a)=(1,5,4,2,6,3)$$
$$V_{(5)}(a)=(1,6,2,5,3,4)$$

它们可由一个置换(1,3,5,6,4,2)的方幂生成。

（2）VINO 加密算法采用运算的中间结果作为参数来控制 VV 置换，以增加变化的不固定因素，无疑给密码破译者增加了分析难度。然而，深入思考就会发现事物的另一面：以不固定因素作为控制参数，当参数是某些特殊情形时，就会相对地减少密码体制的密度，例如，当控制参数 $H$=(0,0,⋯,0)时，对任意输入 1 都有：

$$\mathrm{VV}_H(I)=(0,0,\cdots,0)\text{或}(1,1,\cdots,1)$$

当控制参数 $H$=(0001 0000 0001 0000)时，VV 置换是恒等变换，即对任意输入 $I$ 都有：

$$\mathrm{VV}_H(I)=I$$

## 1.31　MULTI2 加密算法

MULTI2 加密算法是由日立公司的 Takaragi K.、Sasaki R.和 Nakagawa F.于 1989 年 8 月提出的。

MULTI2 加密算法是一个代替-置换密码算法，在代替-置换的变换过程中，每次循环移位数量都是不同的，它使得一个输入比特的变化会影响整个输出比特，所以要求代替-置换的迭代次数会尽可能小。

图 1.31.1 是 MULTI2 加密算法的代替-置换结构，其置换使用通用 32 位处理器的标准机器指令来实现，这使得快速软件实现和小程序存储区域成为可能。

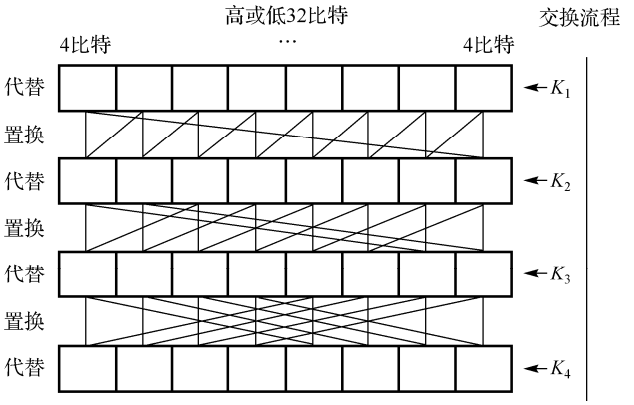



图 1.31.1　MULTI2 加密算法的代替-置换结构

图 1.31.2 是 MULTI2 加密算法的 4 个基本函数 $\pi_1$、$\pi_2$、$\pi_3$、$\pi_4$ 的变化流程，图中，⊕表示按比特模 2 加；+表示按比特模 $2^{32}$ 相加；–表示按比特模 $2^{32}$ 相减；Rot$s$ 表示循环左移 $s$ 比特；∨表示按比特逻辑或。

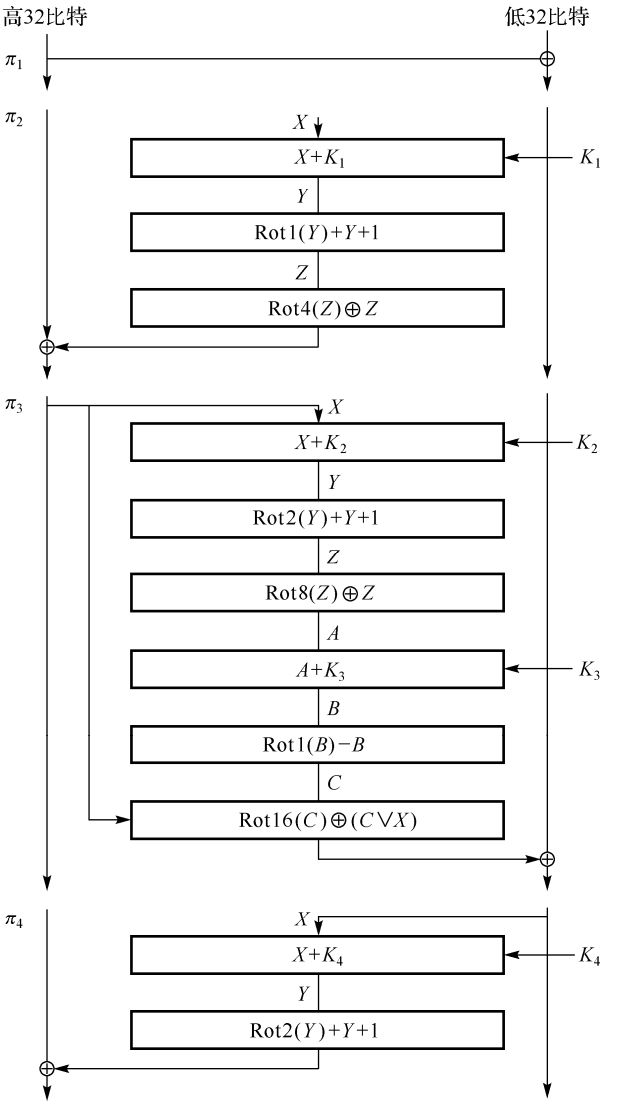



图 1.31.2　MULTI2 加密算法的 4 个基本函数

图 1.31.3 是 $N$ 级 MULTI2 加密算法加密流程，它是为抵抗选择明文攻击而提出的，这里的 1 级由 4 次对合（Involution）组成，$N$ 级则由 $4N$ 次对合组成。

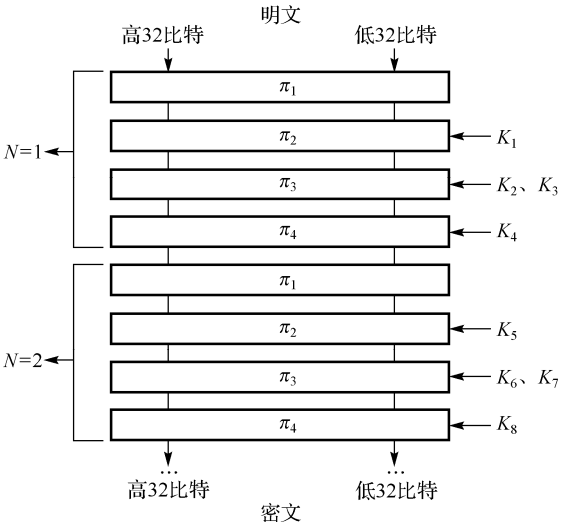



图 1.31.3　$N$ 级 MULTI2 加密算法加密流程

图 1.31.4 是 MULTI2 加密算法加密过程所需的加密密钥 $K_1$、$K_2$、$K_3$…的生成过程，用户的数据密钥长度为 64 比特，$I_1$、$I_2$、$I_3$…是系统密钥。

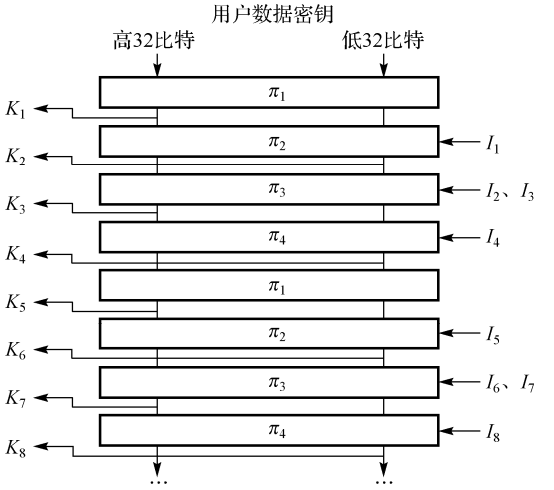



图 1.31.4　MULTI2 加密算法加密过程所需的加密密钥的生成过程

## 1.32　非线性奇偶电路密码体制

非线性奇偶电路（Nonlinear Parity Circuits）是由日本 NTT 的 Kenji Koyama 和 Routo Terada

在1990年提出的构思简洁的分组加密函数，发表于1990年美洲密码年会。它是$\{0,1\}^n$到$\{0,1\}^n$的1-1映射$F(K,A)$，$A$是输入，$A\in\{0,1\}^n$，$K$是密钥，$K\in\{0,1+,-\}^n$。

下面先定义一层的非线性奇偶电路。设输入分组 $A=a_1,a_2,\cdots,a_n$, $a_n\in\{0,1\}^n$；密钥分组 $K=k_1,k_2,\cdots,k_n$, $k_n\in\{0,1+,-\}^n$；输出分组 $B=f(K,A)=b_1,b_2,\cdots,b_n$, $b_n\in\{0,1\}^n$。

### 1．一层非线性奇偶电路

（1）由输入$A$与密钥$K$计算$T$，即

$$T=\sum_{j=1}^{n}t_j \pmod 2$$

式中，

$$t_j=\begin{cases}1, & k_j=0 \text{ 且 } a_j=0 \text{ 或 } k_j=1 \text{ 且 } a_j=1\\ 0, & \text{其他}\end{cases} \qquad (1\text{-}32\text{-}1)$$

$T$实质上是$A$与$K$在对应比特位置上输入比特与密钥比特同号的个数的奇偶性。

（2）输出$B=b_1,b_2,\cdots,b_n$，其中

$$b_j=\begin{cases}a_j, & k_j=1 \text{ 或} k_j=+，\text{当} T=0\pmod 2，\text{即} T\text{为偶时}\\ & \text{或} k_j=-，\text{当} T=0\pmod 2，\text{即} T\text{为奇时}\\ a_j, & \text{其他}\end{cases} \qquad (1\text{-}32\text{-}2)$$

**例1：**

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ |
|---|---|---|---|---|---|---|---|---|---|

输入$A$：

| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| $k_1$ | $k_2$ | $k_3$ | $k_4$ | $k_5$ | $k_6$ | $k_7$ | $k_8$ | $k_9$ | $k_{10}$ |

密钥$K$：

| – | 0 | 1 | – | + | + | 1 | 1 | – | + |
|---|---|---|---|---|---|---|---|---|---|

输出$B$：

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|

先计算$T$，因为$a_2=k_2=0$，$a_3=k_3=1$，$a_7=k_7=1$，所以

$$T=3\equiv 1 \pmod 2$$

根据式（1-32-2），可得$B=0000000111$。

**例2：**

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 输入 $A$： | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 密钥 $K$： | − | 0 | 1 | − | + | + | 1 | 1 | − | + |
| 输出 $B$： | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

计算

$$T=2\equiv 0 \pmod 2$$

根据式（1-32-2）可得到 $B$。

### 2．多层非线性奇偶电路

用 $C(n, d)$表示分组长度为 $n$ 的 $d$ 层奇偶电路；密钥 $K=k_1 \| k_2 \| \cdots \| k_d$，$k_i$ 为第 $i$ 层密钥；输入 $A=a_1,a_2,\cdots,a_n$；单层加密变换记为 $f$；$d$ 层加密函数为

$$F(K,A)=f\{k_d,f\{k_{d-1},[\cdots,f(k_1,A)\cdots]\}\} \tag{1-32-3}$$

即第 1 层输入为 $A$，它与密钥 $k_1$ 作用经层加密函数 $f$ 的输出 $f(k_1,A)$作为第 2 层的输入，…，第 $i$ 层的输出 $f(k_i,\cdot)$作为第 $i$+1 层的输入，直到第 $d$ 层为止。

**例 3：**

$C(n, d)n=10$，$d=3$

输入 $A$：

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 密钥 $k_1$： | | − | 0 | 1 | − | + | + | 1 | 1 | − | + |
| $B_1=f(k_1,A)$： | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| $k_2$： | | + | 1 | 0 | 1 | 1 | + | 0 | − | + | − |
| $B_2=f(k_2,B_1)$： | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| $k_3$： | | − | 0 | 1 | + | + | 0 | − | + | + | − |
| $B_3=f(k_3,B_2)$： | | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

$T=3\equiv 1 \pmod 2$

$T=2\equiv 0 \pmod 2$

$T=0\equiv 0 \pmod 2$

$B_3$ 就是输入 $A$ 经过 3 层非线性奇偶电路变换后的输出。

由式（1-32-3）容易得出脱密变换，即

$$F^{-1}(K,B)=F^{-1}\{K_1,F^{-1}\{K_2,\{\cdots,F^{-1}[K_{d-1},F^{-1}(K_d,B)\cdots]\}\}\}$$

此处 $B$ 为已知的输出，即

$$B=F(K,A)$$

下面讨论一层的脱密函数 $F^{-1}(K,B)$。

设 $K=k_1,k_2,\cdots,k_n$，定义 $B'=b'_1,b'_2,\cdots,b'_n$，其中

$$b'_j=\begin{cases}\overline{b_j}, & k_j=1\\ b_j, & \text{其他}\end{cases}$$

由式（1-32-1）可见，$T=T(K,A)$与 $K$ 中的“+”“–”无关，故 $T(K,B')=T(K,A)$，所以只要按式（1-32-1）计算：

$$T=T(K,B')$$

然后计算：

$$a_j=\begin{cases}b'_j, & k_j=-\text{且}T=1\text{或}k_j=+\text{且}T=0\\ b'_j, & \text{其他}\end{cases}$$

这样就可得到原始输入 $A=a_1,a_2,\cdots,a_n$。

**例 4**：求例 2 的逆。

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 已知输出 $B$： | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 密钥 $K$： | – | 0 | 1 | – | + | + | 1 | 1 | – | + |
| $B'$： | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

求出原始输入 $A$=1001001001。

对于多层非线性奇偶电路 $C(n,d)$，其脱密变换是先求出 $F^{-1}(K_d,B)$，再一层一层地往外脱密，即再求 $F^{-1}[K_{d-1},F^{-1}(K_d)]\cdots$，脱密密钥是 $k_d \parallel k_{d-1} \parallel k_{d-2} \parallel \cdots \parallel k_1$。

### 3．非线性奇偶电路特点及其分析

（1）从上面加密算法来看，运算极为简洁，易用硬件、软件实现，硬件实现的加密速度如表 1.32.1 所示。

表 1.32.1　硬件实现的加密速度

| 分组长 $n$/比特 | 层数 $d$ | 速度/（百万比特/秒） | 所用门电路数/千门 |
|---|---|---|---|
| 64 | 8 | 160 | 3.2 |
| 64 | 16 | 80 | 3.2 |
| 128 | 8 | 300 | 6.4 |
| 128 | 16 | 150 | 6.4 |

（2）非线性。从一层加密变换看，显然有：

$$f(K, A_1 \oplus A_2) \neq f(K, A_1) \oplus f(k, A_2)$$

但如果密钥 $K \in \{-,+\}^n$ 或 $K \in \{0,1\}^n$，则有：

$$f(K,A_1 \oplus A_2)=f(K,A_1) \oplus f(k,A_2)$$

因此在加密之前，首先要检验密钥序列，从而避免以下密钥序列的使用，即当 $d$=1 时，$2 \cdot 2^n/4^n=1/2^{n-1}$；当 $d$>1 时，$2^d \cdot 2^n/4^{dn}=1/2^{dn-d-n}$。

（3）当 $d$ 足够大时，$C(n,d)$符合雪崩准则，即改变输入一个比特，整个输出平均有一半比特改变。

设 $A_1$、$A_2$ 为两个不同的输入，$A_1$ 改变 $i$ 比特成为 $A_2$，即 $A_1$ 与 $A_2$ 的汉明距离为 $i$，输出 $F(k,A_1)$与 $F(k,A_2)$的平均汉明距离为 $H_r(n,d,i)$。这里平均的含义是对所有的汉明距离为 $i$ 的输入对$(A_1,A_2)$与所有的 $4^{nd}$ 密钥而言的。

对一层非线性奇偶电路，有：

$$H_r(n,1,i)=n/4+3/4, \quad i=1$$

$$H_r(n,1,i)=n/4+i/2, \quad i \geq 2$$

设多层非线性奇偶电路 $C(n,d)$是均匀分布的，则有：

$$n/2+(1/2)^d(i-n/2) \leqslant H_r(n,d,i) \leqslant (n+1)/2+(1/2)^d[i-(n+1)/2]$$

当 $d \to \infty$时，$H_r(n,d,i)$收敛在 $n/2$ 与$(n+1)/2$ 之间。

表 1.32.2 是平均输出汉明距离，$n \geq 8$ 的数据是通过对 100 万次实验结果统计得到的。

表 1.32.2　平均输出汉明距离

| | $n$=1 | $n$=2 | $n$=3 | $n$=4 | $n$=6 | $n$=8 | $n$=16 | $n$=32 | $n$=64 |
|---|---|---|---|---|---|---|---|---|---|
| $d$=1 | 1 | 1.2500 | 1.5000 | 1.7500 | 2.2500 | 2.7500 | 4.7500 | 8.7500 | 16.7500 |
| $d$=2 | 1 | 1.3125 | 1.6563 | 2.0156 | 2.7539 | 3.5469 | 6.3750 | 11.7500 | 22.508 |
| $d$=3 | 1 | 1.3281 | 1.7031 | 2.1055 | 3.9502 | 3.9625 | 7.4625 | 14.463 | 28.463 |
| $d$=16 | 1 | 1.331 | 1.7141 | 2.1331 | 3.0474 | 4.0021 | 8.0000 | 16.000 | 32.000 |
| $d$=∞ | 1 | 1.3333 | 1.7142 | 2.1333 | 3.0476 | 4.0023 | 8.0000 | 16.000 | 32.000 |

由表 1.32.2 可见，当 $d$ 适当增大时，$H_r(n,d,i)=n/2$，即 $C(n,d)$的输入和输出满足雪崩准则。同样，密钥与输出也符合雪崩准则，即若密钥改变 1 比特，输出比特平均要改变一半（这里不再详述）。

（4）互补性。定义 $\overline{K}$、$\overline{I}$ 为对 $K$、$I$ 每比特分别求补后的结果。对于密钥定义“+”的补是“−”，“−”的补是“+”。

对于一层非线性奇偶电路，总有下面互补性成立，即 $f(K,I)=f(\overline{K},\overline{I})$ 对所有的密钥 $K$ 与输入 $I$ 都成立。

这是因为在 $K$ 的“0”“1”位置上，$\overline{K} \oplus \overline{I}=K \oplus I$，而 $K$ 与 $I$ 同时取补并不改变奇偶性。

因而在 $K$ 的“+”“–”位置上，$\bar{K}$ 对 $\bar{I}$ 的作用与 $K$ 对 $I$ 的作用是相同的。然而，多层非线性奇偶电路 $C(n,d)$的互补性不一定成立。但不管怎么样，互补性对密码分析是有利的，它至少可以在一定程度上减少对密钥的穷举量。

（5）等价密钥的存在。设 $C(n,d)$的输出 $O=F(K,I)$，对于某一固定的输入，其互异的输出共有 $2^n$ 个，由于 $F$ 是 1-1 映射的，不同的输出必然是由不同的密钥作用所得到的，而互异的密钥 $K$ 共有 $4^{dn}$ 个，因此必有 2 个不同的密钥作用产生相同的输入和输出。由于密钥与输入、输出都是均匀分布的，对于给定的一对（$I_i$, $O_j$），总有不同的 $K_1$、$K_2$ 满足：

$$O_j=F(K_1, I_i)=F(K_2, I_i)$$

称这样的 $K_1$、$K_2$ 为（$I_i$, $O_j$）的等价密钥。进一步讲，对于不同的输入-输出对，其等价密钥的数量都相同。例如，$n$=2、$d$=1 时不同的输入-输出对的等价密钥如表 1.32.3 所示。

表 1.32.3　$n$=2、$d$=1 时不同的输入-输出对的等价密钥

| | (0,0) | (0,1) | (1,0) | (1,1) |
|---|---|---|---|---|
| (0,0) | (0,0)(–,–) | (0,1)(–,+) | (1,0)(+,–) | (1,1)(+,+) |
| | (0,+)(+,0) | (0,–)(+,1) | (–,0)(1,+) | (–,1)(1,–) |
| (0,1) | (0,1)(–,+) | (0,0)(–,–) | (1,1)(+,+) | (1,0)(+,–) |
| | (–,1)(0,–) | (–,0)(0,+) | (+,1)(1,–) | (+,0)(1,+) |
| (1,0) | (1,0)(+,–) | (1,1)(+,+) | (0,0)(–,–) | (0,1)(–,+) |
| | (–,0)(1,–) | (–,1)(1,+) | (0,–)(+,0) | (0,+)(+,1) |
| (1,1) | (1,1)(+,+) | (1,0)(+,–) | (0,1)(–,+) | (0,0)(–,–) |
| | (+,1)(1,+) | (+,0)(1,–) | (–,1)(0,+) | (–,0)(0,–) |

把等价密钥个数记为 $N(n, d)$，显然有：

$$N(n,d)=4^{dn}/2^n=2^{(2d-1)n}$$

由此可见，随着 $d$ 的增加，等价密钥成指数增加，加密时可通过增大 $d$ 和 $n$ 来提高密码的强度（如前文已讨论的密码的完善性和雪崩效应），但这也将增加大量的等价密钥，等价密钥量 $N(n, d)$随 $d$、$n$ 增加的速度如表 1.32.4 所示。

表 1.32.4　等价密钥量 $N(n,d)$随 $d$、$n$ 增加的速度

| | $n$=1 | $n$=2 | $n$=4 |
|---|---|---|---|
| $d$=1 | 2 | 4 | 16 |
| $d$=2 | 8 | 64 | 4096 |
| $d$=3 | 32 | 1024 | 1048576 |

等价密钥的存在增加了破译的可能性，即密码分析者不一定必须求出加密者所用的原始密钥，而只要求出其等价密钥中任意一个就能到达破译的目的。

$C(n, d)$不但有等价密钥，而且还有完全等价密钥。密钥 $K_1$、$K_2$ 为完全等价密钥是指对一

切输入 $I$，都有 $F(K_1,I)=F(K_2,I)$。当 $d>n/2$ 时，它的存在性是明显的，因为当 $k$ 不同时，互异的 $F(K,\cdot)$至多是 $2^n!$个，当 $d>n/2$ 时，密钥总数 $4^{nd}>2^n!$，故必有 $K_1$、$K_2$ 使得 $F(K_1,\cdot)=F(K_2,\cdot)$。

显然：

$$K_1=(K_1,K_2,\cdots,K_n),\qquad K_i\in\{0,1\}$$

$$K_2=(K_1',K_2',\cdots,K_n'),\qquad K_i'\in\{-,+\}\text{ 且 }K_i=0\text{ 时 }K_1'=-\text{；}K_i=1\text{ 时 }K_i'=+$$

是完全等价密钥，这样的完全等价密钥有 $2^{nd}$ 对。对于加密者来说，设计密码时要避免完全等价密钥，以免使密码分析者有可乘之机。

（6）加密所需的密钥比特量大大多于（$2d$ 倍）输入和输出的比特量。上面已说过，$C(n,d)$所用的密钥 $K\in\{0,1,-,+\}^n$，实际在产生密钥 $K$ 时，用二元比特组来代替“0”“1”“–”“+”4 个符号，即

“0　0”→“0”

“0　1”→“1”

“1　0”→“–”

“1　1”→“+”

因此用 $C(n, d)$来加密长度为 $n$ 比特的分组时，需要 $2dn$ 比特的密钥量。也就是说，要以 $2dn$ 比特的随机序列为代价来获得对 $n$ 比特信息的加密。

## 1.33　CRYPTON 加密算法

CRYPTON 加密算法是 AES 的 15 个候选算法之一，它是由韩国的 Chae Hoon Lim 于 1998 年提交的一个分组密码。它的结构和 SQUARE 加密算法完全相同，不属于 Feistel 结构。CRYPTON 加密算法的分组长度为 128 比特，密钥长度可以为 128、160、192、224、256 比特，迭代层数 $r$ 为 12。

CRYPTON 加密算法的结构如图 1.33.1 所示。

将 128 比特分组与第一个层密钥进行模 2 加后，从左到右记成 4 个 32 比特的字 $A=(A_3,A_2,A_1,A_0)$，再将每个字记成 4 字节 $A[i]=(a_{i3},a_{i2},a_{i1},a_{i0})$，这样可将每个分组的 16 字节排成如下所示的 4×4 的方阵。

| | | | | |
|---|---|---|---|---|
| $A_0$ | $a_{03}$ | $a_{02}$ | $a_{01}$ | $a_{00}$ |
| $A_1$ | $a_{13}$ | $a_{12}$ | $a_{11}$ | $a_{10}$ |
| $A_2$ | $a_{23}$ | $a_{22}$ | $a_{21}$ | $a_{20}$ |
| $A_3$ | $a_{33}$ | $a_{23}$ | $a_{31}$ | $a_{30}$ |

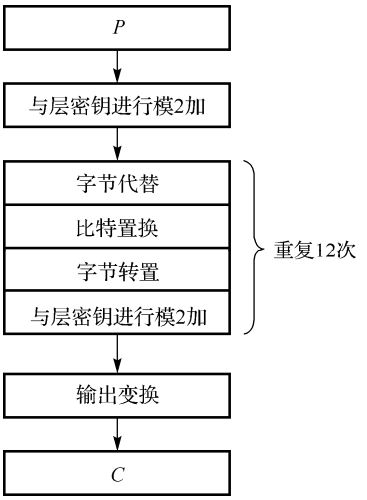



图 1.33.1　CRYPTON 加密算法的结构

CRYPTON 加密算法的层函数 $\rho$ 由 4 个变换组成，即 $\sigma$、$\tau$、$\pi$、$\gamma$。下面依次介绍这 4 个变换。

(1)$\gamma$ 是一个非线性变换，它对方阵中的每个字节进行一次非线性代替，该变换由 2 个 8×8 的 $S$ 盒，即 $S_0$ 和 $S_1$ 组成，$S_0$ 和 $S_1$ 按一定顺序组成 $\gamma$，偶数层和奇数层的 $\gamma$ 不同，将偶数层的 $\gamma$ 记为 $\gamma_e$，将奇数层的 $\gamma$ 记为 $\gamma_o$。

记 $B=\gamma(A)$，$\gamma$ 变换如图 1.33.2 所示。

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A_0$ | $a_{03}$ | $a_{02}$ | $a_{01}$ | $a_{00}$ | $\xrightarrow{\gamma_o}$ | $B_0$ | $S_1(a_{03})$ | $S_0(a_{02})$ | $S_1(a_{01})$ | $S_0(a_{00})$ |
| $A_1$ | $a_{13}$ | $a_{12}$ | $a_{11}$ | $a_{10}$ | | $B_1$ | $S_0(a_{13})$ | $S_1(a_{12})$ | $S_0(a_{11})$ | $S_1(a_{10})$ |
| $A_2$ | $a_{23}$ | $a_{22}$ | $a_{21}$ | $a_{20}$ | | $B_2$ | $S_1(a_{23})$ | $S_0(a_{22})$ | $S_1(a_{21})$ | $S_0(a_{20})$ |
| $A_3$ | $a_{33}$ | $a_{23}$ | $a_{31}$ | $a_{30}$ | | $B_3$ | $S_0(a_{33})$ | $S_1(a_{23})$ | $S_0(a_{31})$ | $S_1(a_{30})$ |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A_0$ | $a_{03}$ | $a_{02}$ | $a_{01}$ | $a_{00}$ | $\xrightarrow{\gamma_e}$ | $B_0$ | $S_0(a_{03})$ | $S_1(a_{02})$ | $S_0(a_{01})$ | $S_1(a_{00})$ |
| $A_1$ | $a_{13}$ | $a_{12}$ | $a_{11}$ | $a_{10}$ | | $B_1$ | $S_1(a_{13})$ | $S_0(a_{12})$ | $S_1(a_{11})$ | $S_0(a_{10})$ |
| $A_2$ | $a_{23}$ | $a_{22}$ | $a_{21}$ | $a_{20}$ | | $B_2$ | $S_0(a_{23})$ | $S_1(a_{22})$ | $S_0(a_{21})$ | $S_1(a_{20})$ |
| $A_3$ | $a_{33}$ | $a_{23}$ | $a_{31}$ | $a_{30}$ | | $B_3$ | $S_1(a_{33})$ | $S_0(a_{23})$ | $S_1(a_{31})$ | $S_0(a_{30})$ |

图 1.33.2　$\gamma$ 变换

$S_0$ 和 $S_1$ 均为三层的 Feistel 结构，如图 1.33.3 所示。从图 1.33.3 中可以看出，$S_0$ 和 $S_1$ 是互逆的，$P_0$、$P_1$ 和 $P_2$ 是 3 个 4×4 的代替表，如表 1.33.1 所示。

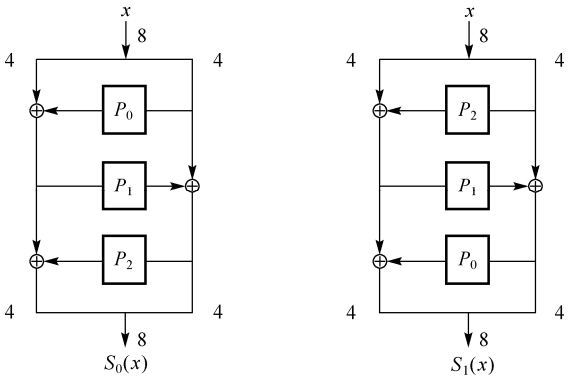



图 1.33.3　CRYPTON 加密算法的 $S_0$ 和 $S_1$

表 1.33.1　$P_0$、$P_1$ 和 $P_2$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_0$ | 15 | 9 | 6 | 8 | 9 | 9 | 4 | 12 | 6 | 2 | 6 | 10 | 1 | 3 | 5 | 15 |
| $P_1$ | 10 | 15 | 4 | 7 | 5 | 2 | 14 | 6 | 9 | 3 | 12 | 8 | 13 | 1 | 11 | 0 |
| $P_2$ | 0 | 4 | 8 | 4 | 2 | 15 | 8 | 13 | 1 | 1 | 15 | 7 | 2 | 11 | 14 | 15 |

（2）$\pi$ 是一个线性变换，它对方阵的行进行比特变换，偶数层和奇数层的 $\pi$ 变换不同，将它们分别记为 $\pi_e$ 和 $\pi_o$。$\pi$ 变换中使用了 4 个 32 比特的掩码字。

$$M_0=m_3 \| m_2 \| m_1 \| m_0=0x3fcff3fc$$
$$M_1=m_0 \| m_3 \| m_2 \| m_1=0xfc3fcff3$$
$$M_2=m_1 \| m_0 \| m_3 \| m_2=0xf3fc3fcf$$
$$M_3=m_2 \| m_1 \| m_0 \| m_3=0xcff3fc3f$$

式中，$m_0$=0xfc, $m_1$=0xf3, $m_2$=0xcf, $m_3$=0x3f。记 $B=\pi_o(A)$，$\pi_o$ 变换为：

$$B_0=(A_0\&M_0)\oplus(A_1\&M_1)\oplus(A_2\&M_2)\oplus(A_3\&M_3)$$
$$B_1=(A_0\&M_1)\oplus(A_1\&M_2)\oplus(A_2\&M_3)\oplus(A_3\&M_0)$$
$$B_2=(A_0\&M_2)\oplus(A_1\&M_3)\oplus(A_2\&M_0)\oplus(A_3\&M_1)$$
$$B_3=(A_0\&M_3)\oplus(A_1\&M_0)\oplus(A_2\&M_1)\oplus(A_3\&M_2)$$

记 $B=\pi_e(A)$，$\pi_e$ 变换为：

$$B_0=(A_0\&M_1)\oplus(A_1\&M_2)\oplus(A_2\&M_3)\oplus(A_3\&M_0)$$
$$B_1=(A_0\&M_2)\oplus(A_1\&M_3)\oplus(A_2\&M_0)\oplus(A_3\&M_1)$$
$$B_2=(A_0\&M_3)\oplus(A_1\&M_0)\oplus(A_2\&M_1)\oplus(A_3\&M_2)$$
$$B_3=(A_0\&M_0)\oplus(A_1\&M_1)\oplus(A_2\&M_2)\oplus(A_3\&M_3)$$

记 $A^i=(a_{3i},a_{2i},a_{1i},a_{0i})'$为 $A$ 的第 $i$ 个字节列，$\pi_i$ 是由 $\pi$ 变换推出第 $i$ 列的比特变换，则 $\pi_e$ 和 $\pi_o$ 变换可以写成：

$$\pi_o(A)=[\pi_3(A^3),\pi_2(A^2),\pi_1(A^1),\pi_0(A^0)]'$$
$$\pi_e(A)=[\pi_0(A^3),\pi_3(A^2),\pi_2(A^1),\pi_1(A^0)]'$$

式中，$\pi_i$（$i$=0,1,2,3）表示 $\pi_o$ 变换中 $M_j$（$j$=0,1,2,3）的第 $i$ 个字节列组成的变换。例如，$B^0=\pi_0(A^0)$ 的定义为：

$$b_{00}=(a_{00}\&m_0)\oplus(a_{10}\&m_1)\oplus(a_{20}\&m_2)\oplus(a_{30}\&m_3)$$
$$b_{10}=(a_{00}\&m_1)\oplus(a_{10}\&m_2)\oplus(a_{20}\&m_3)\oplus(a_{30}\&m_0)$$
$$b_{20}=(a_{00}\&m_2)\oplus(a_{10}\&m_3)\oplus(a_{20}\&m_0)\oplus(a_{30}\&m_1)$$
$$b_{30}=(a_{00}\&m_3)\oplus(a_{10}\&m_0)\oplus(a_{20}\&m_1)\oplus(a_{30}\&m_2)$$

（3）$\tau$ 变换是对 $A$ 进行转置，使得 $b_{ij}=a_{ji}$，$i$、$j$=0,1,2,3。

（4）$\sigma_K(A)$变换是将层密钥与 $A$ 对应的字节进行模 2 加，层密钥的长度等于输入分组的长度。显然：

$$\pi_e^{-1}=\pi_e,\ \pi_o^{-1}=\pi_o,\ \tau^{-1}=\tau,\ \sigma_K^{-1}=\sigma_K,\ \gamma_o^{-1}=\gamma_e,\ \gamma_e^{-1}=\gamma_o$$

因此，奇数层的层变换 $\rho_{oK}(A)=(\sigma_K\tau\pi_o\gamma_o)(A)$，其逆变换为：

$$\rho_{oK}^{-1}(A)=(\gamma_e\pi_o\tau_{\sigma K})(A)$$

偶数层的层变换 $\rho_{eK}(A)=(\sigma_K\tau\pi_e\gamma_e)(A)$，其逆变换为：

$$\rho_{eK}^{-1}(A)=(\gamma_o\pi_e\tau\sigma_K)(A)$$

算法的输出变换为：

$$\phi_e=\tau\pi_e\tau$$

记 $K_e^i$ 为 CRYPTON 加密算法的第 $i$ 个层密钥，则整个加密算法可以用下面的公式表示：

$$E_K=\phi_e\rho_eK_e^r\rho_oK_e^{r-1}\cdots\rho_eK_e^2\rho_eK_e^1\sigma_{Ke}^0$$

记 $\phi_o=\tau\pi_o\tau$，$K_d^i$ 为脱密算法的第 $i$ 个层密钥，如果

$$K_d^{r-i}=\begin{cases}\phi_e(K_e^i), & i=0,2,4,6,8,10,12\\ \phi_o(K_e^i), & i=1,3,5,7,9,11\end{cases}$$

则整个脱密算法可以用下面的公式表示：

$$D_K=\phi_e\rho_{eKd}^r\rho_{oKd}^{r-1}\cdots\rho_{eKd}^2\rho_{eKd}^1\sigma_{Kd}^0$$

推导过程略。

从上面两个公式可以看出，加/脱密算法可以用同一套装置，但加/脱密算法的层密钥生成方法不同。

整个算法共用了 13 个 32 比特的层密钥，加密算法的层密钥生成方法如下：

（1）由密钥 $K$ 生成扩展密钥。首先将 $K$ 扩展成 256 比特，不足 256 比特的左边补“0”；然后将这 256 比特从左到右记成 8 个 32 比特的字 $U[7]$、$U[6]$、$U[5]$、$U[4]$、$U[3]$、$U[2]$、$U[1]$、$U[0]$；接着用这 8 个字和算法中用到的 4 个变换生成 8 个 32 比特的扩展密钥 $E[i]$, $i$=0,1,…,7，具体生成方法如下：

$$(V[3],V[2],V[1],V[0])' = (\tau\gamma_o\sigma_P\pi_o)(U[6],U[4],U[2],U[0])'$$

$$(V[7],V[6],V[5],V[4])' = (\tau\gamma_e\sigma_Q\pi_e)(U[7],U[5],U[3],U[1])'$$

$$T_0 = V[0] \oplus V[1] \oplus V[2] \oplus V[3]$$

$$T_1 = V[4] \oplus V[5] \oplus V[6] \oplus V[7]$$

$$E_e[i] = V[i] \oplus T_1, \quad i=0,1,2,3$$

$$E_e[i] = V[i] \oplus T_0, \quad i=4,5,6,7$$

式中，$P=(P_3,P_2,P_1,P_0)'$和 $Q=(Q_3,Q_2,Q_1,Q_0)'$是 8 个常数，分别由前 8 个奇素数的平方根的小数位数组成，即

$$P_0=2^{32}(\sqrt{3}-1)=0xbb67ae85$$

$$P_1=2^{32}(\sqrt{5}-2)=0x3c6ef372$$

$$P_2=2^{32}(\sqrt{7}-2)=0xa54ff53a$$

$$P_3=2^{32}(\sqrt{11}-3)=0x510e527f$$

$$Q_0=2^{32}(\sqrt{13}-3)=0x9b05688c$$

$$Q_1=2^{32}(\sqrt{17}-4)=0x1f83d9ab$$

$$Q_2=2^{32}(\sqrt{19}-4)=0x5be0cd19$$

$$Q_3=2^{32}(\sqrt{23}-4)=0xcbbb9d5d$$

（2）由扩展密钥生成加密层密钥。记 $K_e^i=(K_e[4i+3],K_e[4i+2],K_e[4i+1],K_e[4i])'$为加密算法的第 $i$ 个层密钥，则 $K_e[i]=E_e[i]$，$i$=0,1,2,…,7，即 $K_e^0$、$K_e^1$ 可由扩展密钥直接得到。

层密钥 $K_e^{2i+1}$ 和 $K_e^{2i+3}$（$i$=0,1,2,3,4,5）可由层密钥 $K_e^{2i}$ 和 $K_e^{2i+1}$ 通过加常数和循环左移位得到，即对于 $i$=0，有

$$K_e[8] = K_e[0] <<< 8,$$

$$K_e[9] = K_e[1] \oplus RC_0,$$

$$K_e[10] = K_e[2] <<< 16,$$

$$K_e[11]=K_e[3]\oplus RC_0,$$
$$K_e[12]=K_e[4]\oplus RC_0,$$
$$K_e[13]=K_e[5]<<<16,$$
$$K_e[14]=K_e[6]\oplus RC_0,$$
$$K_e[15]=K_e[7]<<<24,$$

对于不同的 $i$ 值，$RC_i$ 分别如下：

$$RC_0=0x01010101$$
$$RC_1=0x02020202$$
$$RC_2=0x04040404$$
$$RC_3=0x08080808$$
$$RC_4=0x10101010$$
$$RC_5=0x20202020$$

对于不同的 $i$ 值，层密钥的 4 个 32 比特的字所做的变换如表 1.33.2 所示。

表 1.33.2　加密算法层密钥的 4 个 32 比特的字所做的变换

| $i$ | $K_e^{2i} \to K_e^{2i+2}$ | | | | $K_e^{2i+1} \to K_e^{2i+3}$ | | | |
|---|---|---|---|---|---|---|---|---|
| | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| 0 | $RC_0$ | 16 | $RC_0$ | 8 | 24 | $RC_0$ | 16 | $RC_0$ |
| 1 | 8 | $RC_1$ | 24 | $RC_1$ | $RC_1$ | 16 | $RC_1$ | 8 |
| 2 | $RC_2$ | 24 | $RC_2$ | 16 | 8 | $RC_2$ | 24 | $RC_2$ |
| 3 | 16 | $RC_3$ | 8 | $RC_3$ | $RC_3$ | 24 | $RC_3$ | 16 |
| 4 | $RC_4$ | 8 | $RC_4$ | 24 | 16 | $RC_4$ | 8 | $RC_4$ |
| 5 | 24 | $RC_5$ | 16 | $RC_5$ | — | — | — | — |

脱密算法的层密钥可以简单地由加密算法的层密钥按下面的公式生成。

$$K_d^{r-i}=\begin{cases}\phi_e(K_e^i), & i=0,2,4,6,8,10,12\\ \phi_o(K_e^i), & i=1,3,5,7,9,11\end{cases}$$

考虑到变换 $\phi_e$ 和 $\phi_o$ 的一些特点，可以给出一种更加有效的生成脱密层密钥的方法。由 $\phi=\tau\pi\tau$ 可以推得 $\phi_i=\tau\pi_i\tau$（$i$=0,1,2,3），因此：

$$\phi_o(A)=[\phi_3(A_3),\phi_2(A_2),\phi_1(A_1),\phi_0(A_0)]'$$
$$\phi_e(A)=[\phi_0(A_3),\phi_3(A_2),\phi_2(A_1),\phi_1(A_0)]'$$

$\phi_i$ 具有下列特性：

$$\phi_i(X<<<8k)=\phi_i(X)<<<(32-8k),\qquad k=1,2,3$$
$$\phi_i(X)=\phi_j(X)<<<8,\qquad j=i+1 \bmod 4$$

$$\phi_t(A_j \oplus C)=\phi_t(A_j) \oplus \phi_t(C)$$

因此，脱密算法层密钥的生成方法如下：

（1）由加密算法扩展密钥生成脱密算法扩展密钥。

$$\{E_d[3],E_d[2],E_d[1],E_d[0]\}=49]),\phi_1(K_e[48])\}$$

$$\{E_d[7],E_d[6],E_d[5],E_d[4]\}=\{\phi_3(K_e[47]),\phi_2(K_e[46]),\phi_1(K_e[45]),\phi_0(K_e[44])\}$$

由 $\phi_t$ 的特性可以简化上面的式子，可得：

$$E_d[0]=\phi_3(E_e[0]) \oplus RC_1 \oplus RC_3 \oplus RC_5$$

$$E_d[1]=\phi_0(E_e[1]) \oplus RC_0 \oplus RC_2 \oplus RC_4$$

$$E_d[2]=\phi_1(E_e[2]) \oplus RC_1 \oplus RC_3 \oplus RC_5$$

$$E_d[3]=\phi_2(E_e[3]) \oplus RC_0 \oplus RC_2 \oplus RC_4$$

$$E_d[4]=\phi_3(E_e[4]) \oplus RC_0 \oplus RC_2 \oplus RC_4$$

$$E_d[5]=\phi_3(E_e[5]) \oplus RC_1 \oplus RC_3$$

$$E_d[6]=\phi_3(E_e[6]) \oplus RC_0 \oplus RC_2 \oplus RC_4$$

$$E_d[7]=\phi_1(E_e[7]) \oplus RC_1 \oplus RC_3$$

（2）由扩展密钥生成脱密算法的层密钥过程同生成加密算法的层密钥过程相同，只是常数和移位数不太相同，如表 1.33.3 所示。

表 1.33.3　脱密算法层密钥的 4 个 32 比特的字所做的变换

| $i$ | $K_d^{2i} \to K_d^{2i+2}$ | | | | $K_d^{2i+1} \to K_d^{2i+3}$ | | | |
|---|---|---|---|---|---|---|---|---|
| | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| 0 | 24 | $RC_5$ | 16 | $RC_5$ | 16 | $RC_4$ | 8 | $RC_4$ |
| 1 | $RC_4$ | 8 | $RC_4$ | 24 | $RC_3$ | 24 | $RC_3$ | 16 |
| 2 | 16 | $RC_3$ | 8 | $RC_3$ | 8 | $RC_2$ | 24 | $RC_2$ |
| 3 | $RC_2$ | 24 | $RC_2$ | 16 | $RC_1$ | 16 | $RC_1$ | 8 |
| 4 | 8 | $RC_1$ | 24 | $RC_1$ | 24 | $RC_0$ | 16 | $RC_0$ |
| 5 | $RC_0$ | 16 | $RC_0$ | 8 | — | — | — | — |

## 1.34　DEAL 加密算法

DEAL 加密算法是由 Lar R. Knudsen 于 1998 年 2 月提出的一种分组密码算法，它是一种基于 DES 的层数 $r$ 可变的 Feistel 结构密码，该密码在层函数中采用 DES，形成一个具有 $r$×64 比特的层密钥的、分组长度为 128 比特的分组密码。这里的 $r$×64 比特的层密钥是通过密钥表算法从用户选择密钥中推导出来的。密钥表适用于密钥长度分别为 128、192 和 256 比特的三

种不同情形，建议前两种密钥长度采用 $r$=6，而后一种采用 $r$=8。该算法已被提议编入 ANSI 标准的第 X9.52 条款中，并把它作为 NIST AES 的候选标准。

通常将 128、192 和 256 比特密钥长度的 DEAL 加密算法分别记为 DEAL-128、DEAL-192 和 DEAL-256 加密算法，它们的体制是一样的，只是子密钥的生成方法有区别。下面以 DEAL 加密算法在 ECB 模式中的工作原理为例来进行介绍。

设 $C=E_B(A)$表示使用具有密钥 $B$ 的 DES 加密算法来加密 64 比特 $A$ 的值；再设 $Y=\mathrm{EA}_Z(X)$ 表示使用具有密钥 $Z$ 的 DEAL 加密算法来加密 128 比特 $X$ 的值，用 $X^{\mathrm{L}}$ 和 $X^{\mathrm{R}}$ 分别表示 $X$ 的左半部分和右半部分；

将明文 $P$ 按 128 比特分组，$P=P_1,P_2,\cdots,P_n$。同样，用 $P_j^{\mathrm{L}}$ 和 $P_j^{\mathrm{R}}$ 分别表示 $P_j$ 的左半部分和右半部分。

DEAL 加密算法按如下方式计算密文。假定对 $P_i$ 加密，设 $X_0^{\mathrm{L}}=P_1^{\mathrm{L}}$，$X_0^{\mathrm{R}}=P_1^{\mathrm{R}}$，则

$$X_j^{\mathrm{L}}=E_{\mathrm{RK}j}(X_{j-1}^{\mathrm{L}})\oplus X_{j-1}^{\mathrm{R}},\qquad j=1,\cdots,r$$

$$X_j^{\mathrm{R}}=X_{j-1}^{\mathrm{L}}$$

$P$ 的密文为

$$C_j=E_{\mathrm{AK}}(P_1)=X_r^{\mathrm{L}}\|X_r^{\mathrm{R}}$$

式中：

① $\mathrm{RK}_j$（$j$=1,⋯,$r$）是由用户密钥生成的子密钥，将在下面介绍。

② 对于 DEAL-128 加密算法和 DEAL-192 加密算法，$r$=6；对于 DEAL-256 加密算法，$r$=8。

DEAL 加密算法的 1 圈逻辑框图如图 1.34.1 所示。

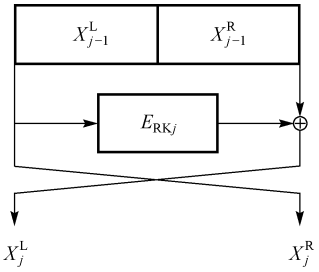



图 1.34.1　DEAL 算法 1 圈逻辑框图

相应地，CBC 工作模式下，$C_i=E_{\mathrm{AK}}(C_{i-1}\oplus P_i)$，其中 $C_0$ 为初始值，$E$ 表示扩展算法。

子密钥生成过程如下：

从 DEAL 加密算法的密钥表中取 $s$ 个密钥（$K_1,\cdots,K_s$，$s$=2,3,4，每个 $K_i$ 均为 64 比特）作

为输入，并回送 $r$ 个子密钥 $RK_j$，取 $K$=0x0123456789abcdef（十六进制数）为一个固定的 DES 密钥。

在 DEAL-128 加密算法中生成的子密钥为：

$$RK_1=E_K(K_1)$$
$$RK_2=E_K(K_2 \oplus RK_1)$$
$$RK_3=E_K(K_1 \oplus <1> \oplus RK_2)$$
$$RK_4=E_K(K_2 \oplus <2> \oplus RK_3)$$
$$RK_5=E_K(K_1 \oplus <4> \oplus RK_4)$$
$$RK_6=E_K(K_2 \oplus <8> \oplus RK_5)$$

式中，<$i$>是一个 64 比特的序数串，该序数串只设置第 $i$–1 比特置为 1，而把其他比特都置为 0，如<1>表示 0x8000000000000000。

DEAL-192 加密算法中生成的子密钥为：

$$RK_1=E_K(K_1)$$
$$RK_2=E_K(K_2 \oplus RK_1)$$
$$RK_3=E_K(K_3 \oplus RK_2)$$
$$RK_4=E_K(K_1 \oplus <1> \oplus RK_3)$$
$$RK_5=E_K(K_5 \oplus <2> \oplus RK_4)$$
$$RK_6=E_K(K_3 \oplus <4> \oplus RK_5)$$

DEAL-256 加密算法中生成的子密钥为：

$$RK_1=E_K(K_1)$$
$$RK_2=E_K(K_2 \oplus RK_1)$$
$$RK_3=E_K(K_3 \oplus RK_2)$$
$$RK_4=E_K(K_4 \oplus RK_3)$$
$$RK_5=E_K(K_1 \oplus <1> \oplus RK_4)$$
$$RK_6=E_K(K_2 \oplus <2> \oplus RK_5)$$
$$RK_7=E_K(K_3 \oplus <4> \oplus RK_6)$$
$$RK_8=E_K(K_4 \oplus <8> \oplus RK_7)$$

DEAL 加密算法的密钥生成不存在明显的弱密钥，且补码特性也不成立。该算法由于码组大，使得匹配密文攻击的可能性变得很小（大约需要 $2^{64}$ 个密文分组）。另外，由于它使用了现成的 DES 加密算法，使能够充分利用现有的 DES 硬件和软件，这在业界很具有吸引力。

## 1.35 DFC 加密算法

DFC（Decorremated Fast Cipher）是法国的 Serge Vaudenay 为响应美国国家标准与技术协会公开征集 AES 而于 1998 年 5 月提出的一候选方案，它是基于 Serge Vaudenay 的抗相关攻击技术设计的，是一种支持分组长度为 128 比特且密钥长度任意（最长可达 256 比特）的分组密码。

### 1．符号说明及常数定义

（1）符号说明。描述该算法时，要用以下一些符号。

0xd43：将比特串按 4 比特一组划分，表示成的十六进制数，0xd43=110101000011。

$s$ 和$|x|_m$：对于比特串 $s=b_1\cdots b_m$，其对应的十进制数 $s=2^{m-1}b_1+2^{m-2}b_2+\cdots+2b_{m-1}+b_m$，$|x|_m$ 是十进制整数 $x$ 的逆操作。例如，0xd43=$\overline{3395}$，$|3395|_{16}$=0xd43。

$s|s'$：表示将比特串 $s$ 和 $s'$连接起来。

$\mathrm{Trunc}_n(s)$：表示将比特串 $s=b_1\cdots b_m$ 的最左 $n$ 比特截断（$m\geqslant n$），$\mathrm{Trunc}_n(s)=b_1\cdots b_n$。

$s\oplus s'$、$s\wedge s'$：分别表示将等长度的比特串 $s$ 和 $s'$按比特进行模 2 加以及按比特与操作。

$\overline{s}$：表示将比特串 $s$ 按比特取反。

（2）常数定义。64 个 32 比特的常数 RT(0),…,RT(63)可按行表示为：

```
0xb7e15162   0x8aed2a6a   0xbf715880   0x9cf4f3c7
0x62e7160f   0x38b4da56   0xa784d904   0x5190cfef
0x324e7738   0x926cfbe5   0xf4bf8d8d   0x8c31d763
0xda06c80a   0xbb1185eb   0x4f7c7b57   0x57f59584
0x90cfd47d   0x7c19bb42   0x158d9554   0xf7b46bce
0xd55c4d79   0xfd5f24d6   0x613c31c3   0x839a2ddf
0x8a9a276b   0xcfbfa1c8   0x77c56284   0xdab79cd4
0xc2b3293d   0x20e9e5ea   0xf02ac60a   0xcc93ed87
0x4422a52e   0xcb238fee   0xe5ab6add   0x835fd1a0
0x753d0a8f   0x78e537d2   0xb95bb79d   0x8dcaec64
0x2c1e9f23   0xb829b5c2   0x780bf387   0x37df8bb3
0x00d01334   0xa0d0bd86   0x45cbfa73   0xa6160ffe
0x393c48cb   0xbbca060f   0x0ff8ec6d   0x31beb5cc
0xeed7f2f0   0xbb088017   0x163bc60d   0xf45a0ecb
0x1bcd289b   0x06cbbfea   0x21ad08e1   0x847f3f73
0x78d56ced   0x94640d6e   0xf0d3d37b   0xe67008e1
```

1 个 64 比特的常数 KD=0x86d1bf275b95241d。

1 个 32 比特的常数 KC=0xab64749a。

3 个 64 比特的常数 $KA_2$、$KA_3$、$KA_4$：

$KA_2$=0xb7e151628aed2a6a

$KA_3$=0xbf7158809cf4f3c7

$KA_4$=0x62e7160f38b4da56

3 个 64 比特的常数 $KB_2$、$KB_3$、$KB_4$：

$KB_2$=0xa784d9045190cfef

$KB_3$=0x324e7738926cfbe5

$KB_4$=0xf4bf8d8d8c31d763

1 个 256 比特的常数 KS，即

KS=0xda06c80abb1185eb4f7c7b5757f59584|0x90cfd47d7c19bb42158d9554f7b46bce

### 2．算法描述

用 $DFC_K$ 表示 DFC 加密算法的加密函数，它利用一个长度最多可达 256 比特的私有密钥 $K$ 作用于一个分组长度为 128 比特的明文。

$DFC_K^{-1}$ 是 $DFC_K$ 的逆函数。

首先通过扩展函数 EF 进行 4 轮 Feistel 算法，然后将私有密钥 $K$ 变成 1024 比特的扩展密钥 EK，即 EK=EF($K$)。加密算法就是利用轮函数 RF 执行一个 8 轮的 Feistel 算法。

（1）置换 CP。置换 CP 是一个在全体 64 比特二进制数构成的集合上的置换，它利用 6 比特的整数作为输入，并以 32 比特二进制数作为输出的替代表 RT 以及常数 KC、KD。

设 $Y=Y_m|Y_r$ 是 CP 的输入，这里 $Y_m$ 和 $Y_r$ 是 2 个 32 比特的二进制数，定义：

$$\mathrm{CP}(Y)=|\{Y_r \oplus \mathrm{RT}[\overline{\mathrm{Trunc}_6(Y_m)}]\}|\overline{[Y_m \oplus (\mathrm{KC})]+\overline{\mathrm{KD}} \bmod 2^{64}}\,|\,64$$

DFC 加密算法的置换 CP 逻辑框图如图 1.35.1 所示。

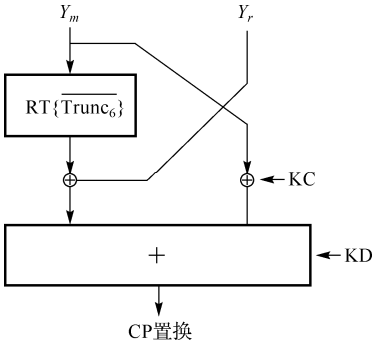



图 1.35.1　DFC 加密算法置换 CP 逻辑框图

（2）轮函数 RF。该函数通过一个 128 比特的参数把一个 64 比特数变换成另一个 64 比特数。128 比特的参数等价于 2 个 64 比特参数，一个为 $a$ 参数，另一个为 $b$ 参数，定义如下：

$$\mathrm{RF}_{a|b}(x)=\mathrm{CP}\{|[(\overline{a}\cdot\overline{x}+\overline{b})\ \bmod\ (2^{64}+13)]\ \bmod\ 2^{64}\ |_{64}\}$$

（3）密钥编排。通过下面的方法可将给定的私有密钥 $K$ 变成 8 个 128 比特的轮密钥 $\mathrm{RK}_1$、$\mathrm{RK}_2$、…、$\mathrm{RK}_8$。整个过程可看成一个函数，记为 EF($K$)。

首先用一个常数 KS 填充 $K$，得到 256 比特的填充密钥 $\mathrm{PK}=\mathrm{Trunc}_{256}(K|\mathrm{KS})$，然后将 PK 分成 8 个 32 比特数，即 $\mathrm{PK}_1,\cdots,\mathrm{PK}_8$，使得 $\mathrm{PK}=\mathrm{PK}_1|\cdots|\mathrm{PK}_8$。定义：

$$\mathrm{OAP}_1=\mathrm{PK}_1|\mathrm{PK}_8$$
$$\mathrm{OBP}_1=\mathrm{PK}_5|\mathrm{PK}_4$$
$$\mathrm{EAP}_1=\mathrm{PK}_2|\mathrm{PK}_7$$
$$\mathrm{EBP}_1=\mathrm{PK}_6|\mathrm{PK}_3$$

对于 $i$=2、3、4，定义：

$$\mathrm{OAP}_i=\mathrm{OAP}_1\oplus\mathrm{KA}_i$$
$$\mathrm{OBP}_i=\mathrm{OBP}_1\oplus\mathrm{KB}_i$$
$$\mathrm{EAP}_i=\mathrm{EAP}_1\oplus\mathrm{KA}_i$$
$$\mathrm{EBP}_i=\mathrm{OBP}_1\oplus\mathrm{KB}_i$$

定义：

$$\mathrm{EF}_1(K)=\mathrm{OAP}_1|\mathrm{OBP}_1|\cdots|\mathrm{OAP}_4|\mathrm{OBP}_4$$
$$\mathrm{EF}_2(K)=\mathrm{EAP}_1|\mathrm{EBP}_1|\cdots|\mathrm{EAP}_4|\mathrm{EBP}_4$$

它们分别确定了一个 4 轮 Testal 型加密函数 $\mathrm{Enc}_{\mathrm{EF}_1(K)}$和 $\mathrm{Enc}_{\mathrm{EF}_2(K)}$（Enc 定义在后面介绍），这两个函数就可定义 PK 序列。设 $\mathrm{RK}_0=|0|_{128}$，有：

$$\mathrm{RK}_i=\begin{cases}\mathrm{Enc}_{\mathrm{EF}_1(K)}(\mathrm{RK}_{i-1}), & i\text{是奇数}\\ \mathrm{Enc}_{\mathrm{EF}_2(K)}(\mathrm{RK}_{i-1}), & i\text{是偶数}\end{cases}$$

也可述为（设 $\mathrm{RK}_{i-1}=\mathrm{RV}_{i,0}|\mathrm{RV}_{i,1}$）：

$$\mathrm{RV}_{i,j+1}=\begin{cases}\mathrm{RF}_{\mathrm{OAP}_j|\mathrm{OBP}_j}(\mathrm{RV}_{i-j})\oplus\mathrm{RV}_{i,j-1}, & i\text{是奇数}\\ \mathrm{RF}_{\mathrm{EAP}_j|\mathrm{EBP}_j}(\mathrm{RV}_{i-j})\oplus\mathrm{RV}_{i,j-1}, & i\text{是偶数}\end{cases}$$

则有：

$$\mathrm{RK}_i=\mathrm{RV}_{i,5}|\mathrm{RV}_{i,4}$$

最后可得：

$$\mathrm{EF}(K)=\mathrm{RK}_1|\mathrm{RK}_2|\cdots|\mathrm{RK}_8$$

（4）加/脱密算法。给定一个 128 比特的明文 PT，将它分成 2 个 64 比特 $R_0$ 和 $R_1$，PT=$R_0|R_1$，私有密钥 $K$ 的扩展密钥 EK=$RK_1|RK_2|\cdots|RK_8$。

通过下面的公式可求得序列 $R_0\cdots R_9$：

$$R_{i+1}=RF_{RK_i}(R_i)\oplus RV_{i-1},\qquad i=1,\cdots,8$$

则密文为：

$$CT=DFC_K(PT)=R_9|R_8$$

DFC 脱密算法的逻辑框图如图 1.35.2 所示。

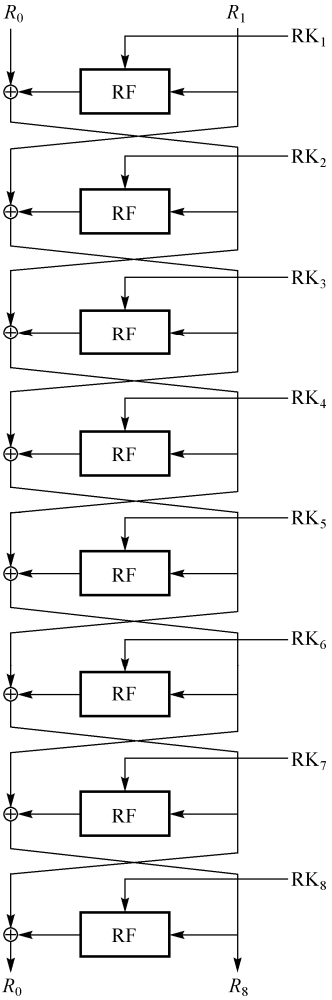



图 1.35.2　DFC 脱密算法的逻辑框图

若一个二进制序列 $s$ 的长度是 128 的整数倍，如 $128r$，则可将其分成 $r$ 个 128 比特的二进制序列，即：

$$s = p_1|p_2|\cdots|p_r$$

对任意 128 比特的序列 $m$，将其分成两个 64 比特 $x_0$ 和 $x_1$，使得 $m=x_0|x_1$，可用下面公式求得序列 $x_0,\cdots,x_{r+1}$，即：

$$x_{i+1}=\mathrm{RF}_{pi}(x_i)\oplus x_{i-1},\quad i=1,\cdots,r$$

定义 $\mathrm{Enc}_s(m)=x_{r+1}|x_r$，即可得到加密函数 $\mathrm{DFC}_K$：

$$\mathrm{DFC}_K=\mathrm{Enc}_{\mathrm{EF}(K)}$$

扩展函数 EF 就是用 Enc 通过 4 轮的 Feistel 定义的。

脱密变换 $\mathrm{DFC}_K^{-1}=\mathrm{Enc}_{\mathrm{revEK}}$，其中

$$\mathrm{revEK}=\mathrm{RK}_8|\mathrm{RK}_7|\cdots|\mathrm{RK}_1$$

DFC 算法的典型特点是，其设计没有使用任何特殊的代数结构，对于差分分析和线性分析，使用少于 $2^{81}$ 次的攻击是难以实现的。

## 1.36 E2 加密算法

E2 是日本电报电话公司于 1998 年 6 提交的、由 Shuuji Shino 等设计的 DES 分组加密算法，其结构是明文先经 IT 变换，再经 12 个含 $F$ 函数的 Feistel 圈，最后经 FT 变换输出密文。

### 1．符号约定

$B$：表示 8 比特元素的向量空间，即 $B=\mathrm{GF}(2^8)$。

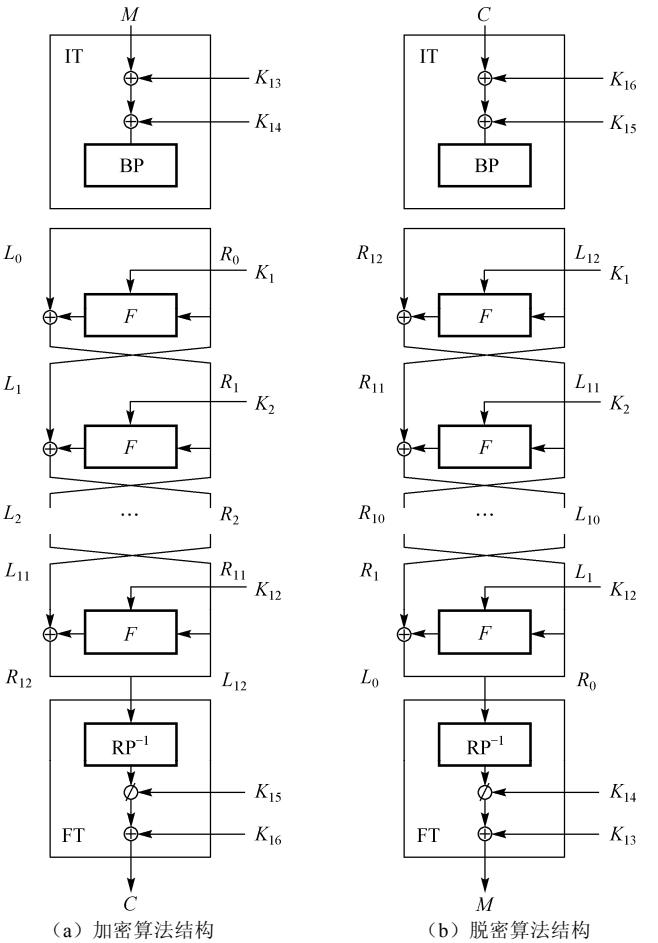
$W$：表示 32 比特元素的向量空间，即 $W=B^4$。

$H$：表示 64 比特（半分组）元素的向量空间，即 $H=B^8$。

$\mathrm{GF}(2^8)$ 的一个元素是指 $\mathrm{GF}(2)[x]$ 中次数小于 8 的一个多项式 $p(x)$，这里的 $\mathrm{GF}(2^8)\cong\mathrm{GF}(2)[x]/r(x)$，$r(x)=x^8+x^4+x^3+x+1$ 是 $\mathrm{GF}(2)[x]$的一个不可约多项式，因此我们可以等同地看待 $B$ 中元素$(a_7\cdots a_0)$与 $\mathrm{GF}(2)^8$ 中的元素 $a_7x^7+a_6x^6+\cdots+a_1x+a_0$。

### 2．加/脱密算法

设 $M$ 是明文（$M\in H^2$），$K$ 是密钥（$K\in H^2$、$H^3$ 或 $H^4$），$C$ 是密文（$C\in H^2$）。密钥 $K$ 在加密前通过密钥编制生成 16 个子密钥 $K_1,K_2,\cdots,K_{16}$，$K_i\in B^{16}$。E2 加/脱密算法如图 1.36.1 所示。加密时，明文 $M$ 先经初始变换 IT，即首先计算：




（a）加密算法结构　　（b）脱密算法结构

图 1.36.1　加/脱密 E2 算法的结构

$$M'=\mathrm{IT}(M,K_{13},K_{14})$$

接着将 $M'$分成等长的两部分，即

$$M'=(L_0,R_0),\quad L_0,R_0\in H$$

对于 $r$=1,2,…,12，计算：

$$R_r=L_{r-1}\oplus F(R_{r-1},K_r)$$

$$L_r=R_{r-1}$$

经 12 个 Feistel 圈后，设 $C'=(R_{12},L_{12})$，最后计算：

$$C=\mathrm{FT}(C',K_{16},K_{15})$$

密文 $C$ 就是加密后结果。

脱密时（子密钥 $K_1,\cdots,K_{16}$ 的生成同上），首先计算：

$$C'=\mathrm{IT}(C,K_{16},K_{15})$$

式中，$C$ 是密文。接着将 $C'$分成等长的两部分，即

$$C'=(R_{12},L_{12}),\quad R_{12},L_{12}\in H$$

对于 $r=12,11,\cdots,1$，计算：

$$L_{r-1}=R_{12}\oplus F(L_r,K_r)$$

$$R_{r-1}=L_r$$

设 $M'=(L_0,R_0)$，最后计算：

$$M=\mathrm{FT}(M',K_{13},K_{14})$$

明文 $M$ 就是脱密后结果。

### 3．函数

（1）IT 变换。IT 变换的定义为：

$$\mathrm{IT}: H^2\times H^2\times H^2\to H^2$$

$$(X,A,B)\to \mathrm{BP}[(X\oplus A)\otimes B]$$

式中，BP 函数（称为字节置换）的定义为：

$$\mathrm{BP}: W^4\to W^4$$

$$(X_1,X_2\cdot X_3,X_4)\to(Y_1,Y_2\cdot Y_3,Y_4),\quad X_i,Y_i\in W;\ i=1,2,3,4$$

设

$$X_i=(x_i^{(1)},x_i^{(2)},x_i^{(3)},x_i^{(4)}),\quad x_i^{(i)}\in B,\ i=1,2,3,4$$

则

$$Y_i=(x_i^{(1)},x_{i+1}^{(2)},x_{i+2}^{(3)},x_{i+3}^{(4)}),\quad i=1,2,3,4$$

式中，$x_{i+4}^{(j)}$ 与 $x_i^{(j)}$ 等同，$i=1,2,3,4$；$j=1,2,3,4$。

$$Y=(Y_1,Y_2\cdot Y_3,Y_4)$$

由此可得出：

$$\mathrm{BP}^{-1}:\ W^4\to W^4$$

$$(Y_1,Y_2\cdot Y_3,Y_4)\to(X_1,X_2\cdot X_3,X_4),\ Y_i,X_i\in W;\ i=1,2,3,4$$

设

$$Y_i=(y_i^{(1)},y_i^{(2)},y_i^{(3)},y_i^{(4)}),\quad y_i^{(j)}\in B$$

则

$$X_i=(y_i^{(1)},y_{i-1}^{(2)},y_{i-2}^{(3)},y_{i-3}^{(4)}),\quad i=1,2,3,4$$

式中，$y_{i-4}^{(j)}$ 与 $y_i^{(j)}$ 等同，$i$=1,2,3,4；$j$=1,2,3,4。

E2 加密算法 BP 函数的逻辑框图如图 1.36.2 所示。

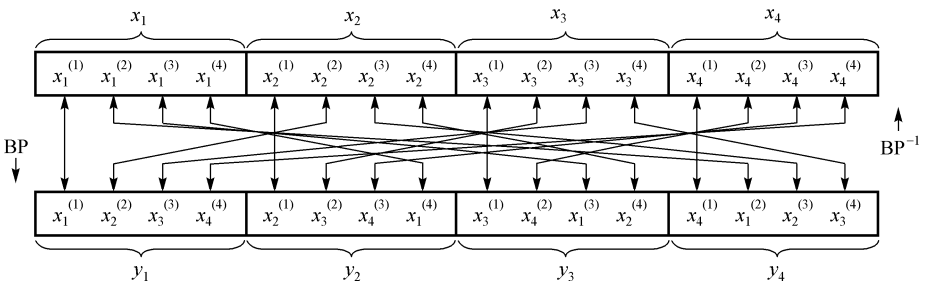



图 1.36.2　E2 加密算法 BP 函数的逻辑框图

二元运算⊗的定义为：

$$H^2\times H^2\to H^2$$
$$Y=X\otimes B,\quad Y,X,B\in H^2$$

设：

$$X=(x_1,x_2\cdot x_3,x_4),\quad x_i^{(j)}\in W,\ i=1,2,3,4$$
$$B=(b_1,b_2\cdot b_3,b_4),\quad b_i\in W,\ i=1,2,3,4$$

则有：

$$y_i=x_i(b_i\vee 1)\bmod 2^{32}Z,\ i=1,2,3,4$$
$$Y=(y_1,y_2\cdot y_3,y_4)$$

式中，$\vee 1$ 表示用 $1\in 2^{32}Z$ 进行逐比特逻辑或运算。

（2）FT 变换。FT 变换是最后的变换，定义如下：

$$\text{FT}:\ H^2\times H^2\times H^2\to H^2$$
$$(X,A,B)\to(\text{BP}^{-1}(X)\varnothing(B)\oplus A$$

式中，$\text{BP}^{-1}$ 如上定义，二元运算∅定义为：

$$X=Y\varnothing B,\ X,Y,B\in H^2$$

设：

$$Y=(y_1,y_2\cdot y_3,y_4),\quad y_i\in W,\ i=1,2,3,4$$
$$B=(b_1,b_2\cdot b_3,b_4),\quad b_i\in W,\ i=1,2,3,4$$

则有：

$$x_i=y_i(b_i\vee 1)^{-1} \bmod 2^{32}Z,\quad i=1,2,3,4$$

$$X=(x_1,x_2\cdot x_3,x_4)$$

式中，$\vee 1$ 表示用 $1\in 2^{32}Z$ 进行逐比特逻辑或运算。

（3）$F$ 函数。$F$ 函数是 E2 加密算法的核心，其定义为：

$$F:\ H\times H^2\times H$$

$$(X,(K^{(1)},K^{(2)}))\rightarrow H^2Y=\mathrm{BRL}\{S\{P[S(X\oplus K^{(1)})]\oplus K^{(2)}\}\}$$

式中，$S$ 函数的定义为：

$$S:\ H\rightarrow H$$

$$(X_1,\cdots,X_8)\rightarrow S(X_1),S(X_2),\cdots,S(X_8)$$

这里 $S$ 盒的定义为：

$$S:\ B\rightarrow B$$

$$x\rightarrow \mathrm{Affine}(\mathrm{Power}(x,127),97,225)$$

式中，$\mathrm{Power}(x,e)=x^e$ 是在 $\mathrm{GF}(2^8)$中运算的；Affine 为仿射函数。

$$\mathrm{Affine}(y,a,b)=ay+b \ (\mathrm{mod}\ 2^8Z)$$

式中，$\mathrm{GF}(2^8)$、$\mathrm{GF}(2)[x]/r(x)$、$Z/2^8Z$ 都是等同的。

由以上面的公式可知，$S$ 盒可由表 1.36.1 给出，即

$$s(0)=225,\quad s(1)=66,\quad s(16)=204,\cdots,\quad s(255)=42$$

表 1.36.1　E2 加密算法的 $S$ 盒

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 225 | 66 | 62 | 129 | 78 | 23 | 158 | 253 | 180 | 63 | 44 | 218 | 49 | 30 | 224 | 65 |
| 204 | 243 | 130 | 125 | 124 | 18 | 142 | 187 | 228 | 88 | 21 | 213 | 111 | 233 | 76 | 75 |
| 53 | 123 | 90 | 154 | 144 | 69 | 188 | 248 | 121 | 124 | 27 | 136 | 2 | 171 | 207 | 100 |
| 9 | 12 | 240 | 1 | 164 | 176 | 246 | 147 | 67 | 99 | 134 | 220 | 17 | 165 | 131 | 139 |
| 201 | 208 | 25 | 149 | 106 | 161 | 92 | 36 | 110 | 80 | 33 | 128 | 47 | 231 | 83 | 15 |
| 145 | 34 | 4 | 237 | 166 | 72 | 73 | 103 | 236 | 247 | 192 | 57 | 206 | 242 | 45 | 190 |
| 93 | 28 | 227 | 135 | 7 | 13 | 122 | 244 | 251 | 50 | 245 | 140 | 219 | 143 | 37 | 150 |
| 168 | 234 | 205 | 51 | 101 | 84 | 6 | 141 | 137 | 10 | 94 | 217 | 22 | 14 | 113 | 108 |
| 11 | 255 | 96 | 210 | 46 | 211 | 200 | 85 | 194 | 35 | 183 | 116 | 226 | 155 | 223 | 119 |
| 43 | 185 | 60 | 98 | 19 | 229 | 148 | 52 | 177 | 39 | 132 | 159 | 215 | 81 | 0 | 97 |
| 173 | 133 | 114 | 3 | 8 | 64 | 239 | 104 | 254 | 151 | 31 | 222 | 175 | 102 | 232 | 184 |
| 174 | 189 | 179 | 235 | 198 | 107 | 71 | 169 | 216 | 167 | 114 | 238 | 29 | 126 | 170 | 182 |
| 117 | 203 | 212 | 48 | 105 | 32 | 127 | 55 | 91 | 157 | 120 | 163 | 241 | 118 | 250 | 5 |
| 61 | 58 | 68 | 87 | 59 | 202 | 199 | 138 | 24 | 70 | 156 | 191 | 186 | 56 | 86 | 26 |
| 146 | 77 | 38 | 41 | 162 | 152 | 16 | 153 | 112 | 160 | 197 | 40 | 193 | 109 | 20 | 172 |
| 249 | 95 | 79 | 196 | 195 | 209 | 252 | 221 | 178 | 89 | 230 | 181 | 54 | 82 | 74 | 42 |

（4）$P$ 函数。$F$ 函数中包含的 $P$ 函数定义如下：

$$P:\ H \to H$$

$$\begin{pmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_8 \end{pmatrix} \to \begin{pmatrix} Z_1' \\ Z_2' \\ \vdots \\ Z_8' \end{pmatrix} = P \begin{pmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_8 \end{pmatrix}$$

式中，$P$ 的取值为：

$$P = \begin{pmatrix} 0111\ 1110 \\ 1011\ 0111 \\ 1101\ 1011 \\ 1110\ 1101 \\ 1101\ 1100 \\ 1110\ 0110 \\ 0111\ 0011 \\ 1011\ 1001 \end{pmatrix}$$

（5）BRL 函数。$F$ 函数中包含的 BRL 函数为字节左循环移位函数，其定义为：

$$\text{BRL}:\ H \to H$$

$$(b_1, b_2, \cdots, b_8) \to (b_2, \cdots, b_8, b_1),\ b_i \in B$$

包含 $S$、$P$、BRL 函数的 $F$ 函数如图 1.36.3 所示。

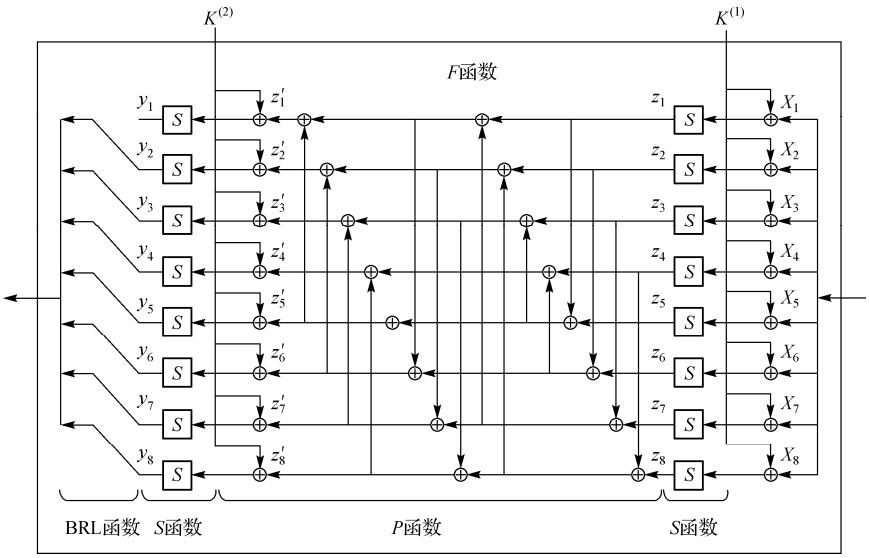



图 1.36.3　包含 $S$、$P$、BRL 函数的 $F$ 函数

### 4．密钥编制

当密钥 $K$ 为 256 比特时，设

$$K=(K_1,K_2,K_3,K_4),\quad K_i\in H,\ i=1,2,3,4$$

将 $K$ 看成输入，利用 $G$ 函数和 $S$ 函数生成子密钥 $k_i\in B^{16}$，$i=1,2,\cdots,16$，其生成过程如下：

令 $V_{-1}$=0x0123456789abcdef，则

$$[L_0,(Y_0,V_0)]=G(K_1,V_{-1})$$

$$[L_{i+1},(Y_{i+1},V_{i+1})]=G(Y_i,V_i),\quad i=0,1,\cdots,7$$

$$(l_{4i},l_{4i+1},l_{4i+2},l_{4i+3})=L_{i+1},\quad i=0,1,\cdots,7$$

$$t_i^{(0)},t_i^{(1)},\cdots,t_i^{(7)}=l_i,\quad i=0,1,2,\cdots,31$$

$$k_{i+1}=(t_{0+(i \bmod 2)}^{\lfloor (i/2)\rfloor},t_{2+(i \bmod 2)}^{\lfloor (i/2)\rfloor},\cdots,t_{30+(i \bmod 2)}^{\lfloor (i/2)\rfloor}),\quad i=0,1,\cdots,15$$

式中，$L_i,Y_i\in H^4$、$l_i,v_i\in H$、$t_i^{(j)}\in B$。

当密钥 $K$ 为 128 比特时，取 $K_3=S\{S[S(V_{-1})]\}$，$K_4=S\{S\{S[S(V_{-1})]\}\}$，显然 $K_3,K_4$ 为常数。当密钥 $K$ 为 192 比特时，取 $K_4=S\{S\{S[S(V_{-1})]\}\}$。

在密钥编制中所用的 $G$ 函数的定义为：

$$G:\ H^4\times H\to H^4\times(H^4\times H)$$

$$[(X_1,X_2\cdot X_3,X_4),U_0]\to\{(U_1,U_2,U_3,U_4),[(Y_1,Y_2,Y_3,Y_4),V]\}$$

式中，

$$Y_i=f(X_i),\quad i=1,2,3,4$$

$$U_i=f(U_{i-1})\oplus Y_i,\quad i=1,2,3,4$$

$$V=U_4$$

E2 加密算法密钥编制中所用的 $G$ 函数的逻辑框图如图 1.36.4 所示，$f$ 函数的定义为：

$$f:\ H\to H$$

$$X\to P[S(X)]$$

即

$$f(X)=P(S(X))$$

这里的 $P$ 函数与 $S$ 函数前面已给出。

### 5．E2 加密算法的测试数据

下面的数据都是以十六进制数的形式给出的。

情况 1：密钥长度为 128 比特。

$K$=00000000000000000000000000000000

$M$=00000000000000000000000000000000

$C$=c2883490b9d9d5e5a03f216edb815fff

情况 2：密钥长度为 192 比特。

$K$=000000000000000000000000000000000000000000000000

$M$=00000000000000000000000000000000

$C$=882f80269d3c146d6ebb9addc4715b4c

情况 3：密钥长度为 256 比特。

$K$=0000000000000000000000000000000000000000000000000000000000000000

$M$=00000000000000000000000000000000

$C$=5002cb8cd878f26fbab9f52e6c96501e

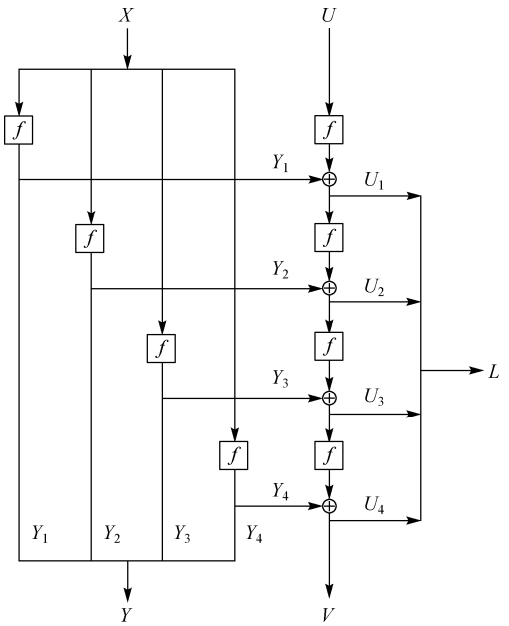



图 1.36.4　E2 加密算法密钥编制中所用的 $G$ 函数的逻辑框图

## 1.37　FROG 加密算法

由 Dianelos Georgoudis 提出的 FROG 加密算法是一种非正规结构的新型分组密码算法，其基本思路是用一种私有的内部密钥来隐蔽大多数计算过程。

FROG 加密算法允许用户密钥的长度为 5～125 字节（即 40～1000 比特），分组大小在 8～

128 字节之间。该算法按字节操作，无比特运算，因此各项参数均是 8 比特的整数倍。图 1.37.1 是 FROG 加/脱密算法的整体结构。

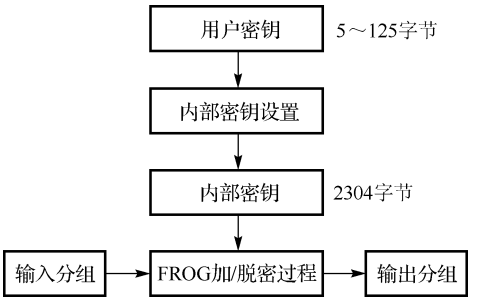



图 1.37.1　FROG 加/脱密算法的整体结构

### 1.37.1　FROG 算法加/脱密过程

FROG 算法使用 8 次迭代，每次迭代都使用内部密钥（internKey）中的一个记录。内部密钥是一个具有 8 个记录的数据结构，每个记录由 3 个数组组成：用一个 16 字节的数组（xorBuf）对分组字节进行初始的模 2 加运算；用一个 256 字节的数组（substPermu）表示字节值的一个代替表；最后用一个 16 字节的数组（bombPermu）中的每一个字节指向分组内不同字节的位置。每次迭代依次移动分组（从最低有效字节到最高有效字节）的字节数，并在每个字节上执行 4 步基本运算，前两步完成混乱，后两步完成扩散。

第 1 步：分组的下一字节与 xorBuf 数组的下一字节进行模 2 加运算。

第 2 步：用第 1 步的结果所指示的代替表（subst Perum）中的字节替代第 1 步中计算出的字节。

第 3 步：修改分组中下一字节中的值，方法是将第 2 步中计算的字节与其进行模 2 加运算。当到达分组的最末字节时，将分组的最低有效字节看成下一字节。

第 4 步：使用 bombPermu 数组的下一字节的内容作为分组字节的序号，并修改该序号字节的值，方法是将第 2 步中计算的字节同其进行模 2 加运算。

在每个字节上进行的 4 步基本运算如图 1.37.2 所示。

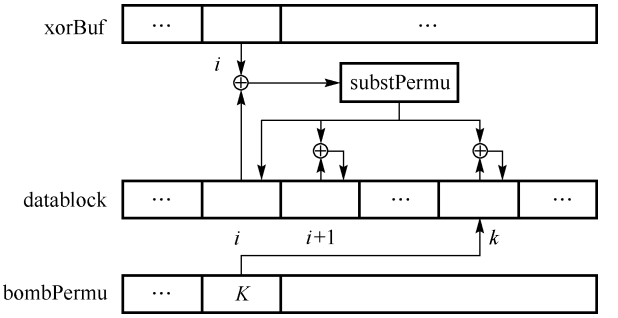



图 1.37.2　在每个字节上进行的 4 步基本运算

加密过程可用下面的伪码来表示，其中的“<=”指赋值运算，blockSize 是指被加密分组的字节个数。

```
Procedure FROGencrytp (plainText,cipherText,internKey)
// convert plainText into cipherText
copy plainText into cipherText
for each of the eight records in internKey do
  begin
  //xorBuf,substPermu,bombPermu denote the fields of the current record
  for each byte in cipherText do [I <= 0 to blockSize-1]
    begin
    cipherText [I] <= cipherText [I] XOR xorBuf [I]
    cipherText [I] <= substPermu [ cipherText[I]]
    If  I<blockSize-1
      then cipherText [I+1] <= cipherText[I+1] XOR cipherText[I]
      else cipherText [0] <= cipherText[0] XOR cipherText[I]
    K <= bombPermu[I]
    cipherText [K] <= cipherText [K] XOR cipherText[I]
    end
  end
end Procedure
```

脱密过程是使用内部密钥（internKey）以反方向对密文进行运算而实现的，伪代码如下：

```
Procedure FROGdecrypt(cipherText,plainText,internKey)
// convert cipherText into plainText
copy cipherText into plainText
for each of the eight records in internKey traversed in opposite direction do
  begin
  //xorBuf,substPermu,bombPermu denote the fields of the current record
  for each byte in plainText do [I <= blockSize-1 down to 0]
    begin
    K <= bombPermu[I]
    plainText[K] <= plainText[K] XOR plainText[I]
    If  I<blockSize-1
      then plainText[I+1] <= plainText[I+1] XOR plainText[I]
      else plainText[0] <= plainText[0] XOR plainText[I]
    plainText[I] <= substPermu [plainText[I]]
    plainText[I] <= plainText[I] XOR xorBuf[I]
    end
  end
end Procedure
```

以上 bombPermu 域用来迅速地扩散整个分组输入变换，为了尽可能快地扩散，该域要满足三个条件：

（1）bombPermu 必须是一个置换。置换的生成由过程 makePermutation 给定：

```
Procedure makePermutation(input,lastElem)
// inputs an array of lastElem+1 byte and conrerts it into a permutation
declare array use of lastElem+1 byte
for I<0 to lastElem do use[I] <= I
last <= lastElem
index <= 0
for all bytes of input [I <= 0 to lastElem-1] do
   begin
   index <= (index+input[I]) mod (last+1)
   input[I] <= use[index]
   if index(last then remove element pointed by index from the use array)
   last <= last-1
   if index>last then index <= 0
   end
   input[lastElem] <= use[0]
end Procedure
```

（2）置换的周期长度应当与明/密文分组大小相等。

（3）置换中的空元素指向其下一个元素的位置。

执行后两个条件的过程称为 Validate，伪代码如下：

```
Procedure Validate(bombpermu)
// make certain that bombPermu has a cycle lenth of blockSize
declare used array of blockSize bytes
fill array used with false
index <= 0
for all but last element in bombPerm [I <= 0 to blockSize-2] do
   begin
   If bombPermu [index]=0 then    //short cycle found
   begin
   K <= index
   repeat K <= (K+1) mod blockSize until NOT used[k] //first position not yet traversed
   bombPermu[index] <= K
   L <= K
   while bombPermu [L] not equal K do L(bombPermu[L])  //find who points to K
   bombPermu[L] <= 0   //now cyclas are merged
   end
   used [index] <= true
   index <= bombPermu [index]   // continue searching
   end
// now make certain that no element of bombpermu points to next element
for all elements in bombpermu [I <= 0 to blockSize-1] do
    if bompermu[I] equal (I+1) mod blockSize
      then bombPermu[I] <= (I+2) mod  blockSize
end Procedure
```

### 1.37.2　密钥设置

FROG 加/脱密算法的密钥设置过程是先建立一个简单的内部密钥，然后使用 FROG 的 CBC 加密模块生成所需的内部密钥，如图 1.37.3 所示。

密钥设置主要使用了两个过程：makeInternalKey 和 haskKey。第一个过程是将一个 2304 字节的无固定结构的随机数组转换成一个有效的内部密钥。具体算法的伪代码如下：

```
Procedure  makeInternalKey (internKey)
    // conrert internKey into a valid FROG internKey
    for each of the eight records in internKey do
    begin //xorBuf,substPermu,bombPermu denote the current records fields
    makePermutation of 256 byte (substPermu)
    if internKey is for decryption then invert (substPermu)
    makePermutation of blockSize bytes (bombPermu)
    Validate (bombPermu)
    end
end Procedure
```

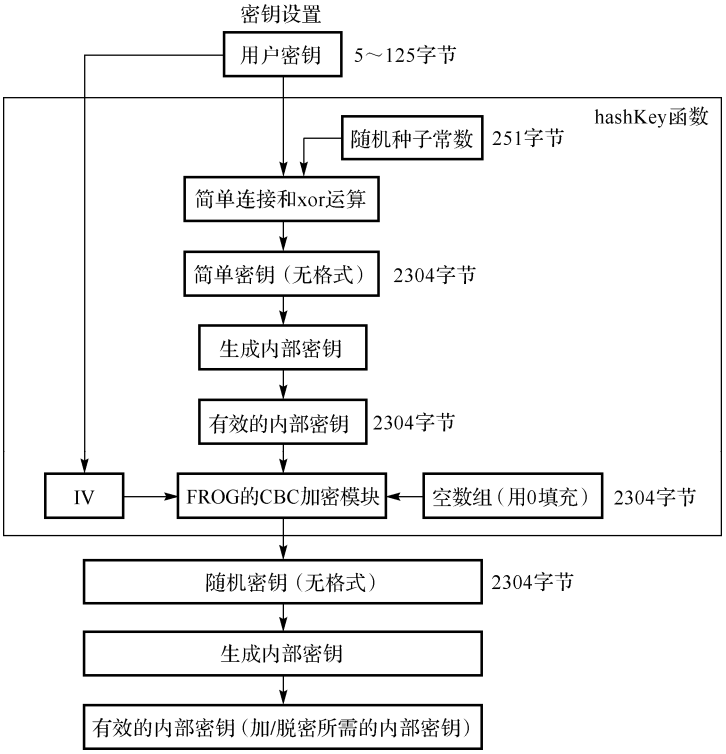



图 1.37.3　FROG 加/脱密算法的密钥设置过程

第二个过程(hashKey)将一个大小为5～125字节的数组转换成不丢失任何熵的一个2304字节的长数组。具体的伪代码如下：

```
Procedure  haskKey(userKey, keyLen, randomKey)
// hash userKey into a randomKey array of internKeySize byte(normal 2304)
//keyLen is the user key longth in byte
//
//Step 1: create simpleKey
//
declare simpleKey array of internKeySize bytes
S <= 0
K <= 0
for each byte in simpleKey do [I <= 0 to internKeySize-1]
   begin
   simpleKey[I] <= randomSeed[S] XOR userKey[K]
   if S<250 then S <= S+1 else S <= 0
   if K<keyLen-1 then K <= K+1 else K <= 0
   end
//convert simpleKey into valid FROG internKey
makeInternalKey for encryption (simpleKey)
//
//step 2: Create buffer to be used subseguently as IV
//
declare buffer array of blocdSize byte and initialize with zeros
last <= keyLen-1
if last greater or equal blockSize then last(blockSize-1)
for I <= 0 to last do buffer[I] <= buffer[I] XOR userKey[I]
buffer[0] <= buffer[0] XOR keyLen
//
//Step 3: call FROG encryption in CBC mode and produce randomKey
//
I <= 0
repeat
   begin
   FROGencrypt (buffer, simpleKey, buffer)
   size <= internKeySize-I
   if size>blockSize then size(blockSize)
   copy size byte of buffer into randomKey [I]
   I <= I+size
   end
   until I equal internKeySize
end Procedure
```

FROG 加密算法的主要优点是设计简单，运算速度快，密钥和分组长度可变等；缺点是在某些应用场合需要较长的设置时间才能生效。

## 1.38　HPC 加密算法

1998 年 6 月，Rich Schroeppel 提出了可接收任何密钥长度、任何分组大小的 HPC（Hasty Pudding Cipher）加密算法。实际上，HPC 加密算法使用 5 个不同的子密码，通过分组的大小可控制采用哪一个子密钥，这 5 个子密钥如表 1.38.1 所示。

表 1.38.1　HPC 加密算法使用的 5 个子密钥

| 序　　号 | 子密钥名 | 分组长度/比特 |
|---|---|---|
| 1 | HPC-Tiny | 0～35 |
| 2 | HPC-Short | 36～64 |
| 3 | HPC-Medium | 65～128 |
| 4 | HPC-Long | 129～512 |
| 5 | HPC-Extended | ≥513 |

### 1.38.1　符号、约定及常数

使用标准 C 语言的操作符，如“=”表示将结果存入左边的变量中；所有的数均为 64 比特的无符号数，加法、减法和乘法都是模 $2^{64}$ 运算；对于无符号数的移位操作，用 0 填充移空的比特位。

比特记数约定：通常以 63 位为一个字的最左边的比特，0 位为最右边的比特。若用一个 ASCII 字符串为密钥，这些字符将从右至左存放在一个 64 比特的数组中，不足时填充 0。例如，包含 9 个字符的字符串“ABCDEFGHI”在内存中表示为：

```
word 0
01001000 01000111 01000110 01000101 01000100 01000011 01000010 01000001
word 1
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01001001
```

HPC 加密算法中十六进制数采用不同的约定：从左到右填满整字，仅对数组末端零碎字（不是 64 比特的字）进行适当调整，如上所示的 72 比特的数可用十六进制表示为 0x484746454443424149。

字约定：HPC 加密算法接收的数据和密钥不一定是 64 比特的整数倍，但均用 64 比特无符号整型数组来表示，其比特数目是一个独立的参数。

大数约定：HPC 加密算法可以加密长度大于 64 比特的整数。这种大数的放置规则是：大数的 64 比特数字位于数组中 word 0 的低 64 比特中，随后 64 比特位于 word 1 中，…，依次类推。如果末端有一个零碎字，其比特进行适当调整。

HPC 中还使用了如下一些内部“随机数”：

```
PI19=3141592653589793238
E19 =2718281828459045235
R220=14142135623730950488
```

### 1.38.2 密钥设置

#### 1．密钥扩展（KX）表

每个子密码都有一个 KX 表，它是由通过用户密钥生成的 256 个 64 比特的字构成的。所有 KX 表均用同一个算法，仅改变初始值。在每个 KX 表的后面再连接最初 30 个字的副本。

建立 KX 表所需的参数为：

- 子密钥序号（从 1 到 5，1 表示 HPC-Tiny，见表 1.38.1）。
- 用户密钥的比特数。
- 一个指向包含用户密钥数组的指针。

若用户密钥的长度不是 64 比特的整数倍，需要在数组最后一个字中适当调整部分比特。KX 表的数组的头三个字初始化为：

KX[0]=PI19+子密钥序号

KX[1]=E19×密钥长度

KX[2]=R220 向左旋转子密钥序号的比特数

余下 253 个字通过下式进行填充：

$$KX[i]=KX[i-1]+(KX[i-2]\oplus KX[i-3]>>23\oplus KX[i-3]<<41)$$

将用户密钥异或到 KX 表的数组中，密钥的字 0（word 0）异或到 KX 数组的字 0（word 1）中，以此类推。如果密钥不是 64 比特的整数倍，则最后一个字在异或前用 0 填充未使用的高比特。当密钥的末端字异或到 KX 表的数组中时，运行如下描述的混乱函数（Stirring Function）以搅乱比特。

对于很长的密钥，将密钥的 128 个字异或到已搅乱的 KX 表的数组中时，混乱函数要至少运行一次，使密钥长度为 0。

混乱函数的目的是使 KX 表的数组伪随机化，使得任一个比特均会影响其他比特。该函数对 KX 数组进行几步变换（默认为三步变换），就改变每个字。在后备选项中，将产生附加

步数，附加步数是参数 BACKUP 和 BACKUPSUPCIPHER[0]的总和；在非后备选项时，这两个变量均为 0。

混乱函数有 8 个内部状态变量，称为 $s_0 \cdots s_7$。在对 KX 表的数组进行第一步变换前，先用 KX 表的数组中最后 8 个值对它们初始化，即

$$s_0 = \text{KX}[248], \cdots, s_7 = \text{KX}[255]$$

KX 表中数组的第一步变换是指 KX 表中数组的每个字同状态变量混乱。各个字的混乱函数用下述伪代码表示：

```
s0 ^= (KX[i] ^ KX[(i+83)&255]) + KX[s0&255]
s1 += s0
s3 ^= s2
s5 -= s4
s7 ^= s6
s3 += s0>>13
s4 ^= s1<<11
s5 ^= s3<<(s1&31)
s6 += s2>>17
s7 |= s3+s4
s2 -= s5
s0 -= s6^i
s1 ^= s5 + PI19
s2 += s7>>j
s2 ^= s1
s4 -= s3
s6 ^= s5
s0 += s7
KX[i] = s2+s6
```

注：$i$ 是被混乱的 KX 表中数组的索引，值为 0～255；$j$ 是步数，第一步的 $j$=0。

完成密钥扩展：将整个密钥异或到 KX 表中数组中，并进行混乱后，数组的前 30 个字被复制到末端。

$$\text{KX}[256]=\text{KX}[0],\text{KX}[257]=\text{KX}[1],\cdots,\text{KX}[285]=\text{KX}[29]$$

### 2．二级密钥 SPICE

SPICE 是 512 比特的二级密钥，存放在 8 个字（每个字为 64 比特）的数组中，默认均为 0。该密钥可根据设计人员的选择，由日期、文件名或磁盘块数等参数自动设置。由于二级密钥便于修改，对每个加密数据块而言，都应改变二级密钥，这可以使主密钥有一个较长的生命期。需要注意的是，在加密时可给二级密钥赋任何值，但在脱密时必须使用匹配值。

### 3．后备选项

HPC 加密算法包括一个后备选项，可随时调用。该选项有一个全局变量 BACKUP 和一个后备变量数组 BACKUPSUBCIPHER[]。数组元素是针对每个子密钥的数，由其确定每个子密钥要做的工作量。在不使用后备选项时，所有后备参数均为 0，只有当这些参数设为正值时，才能使用后备选项。每个子密钥将其后备变量数组与全局变量相加来确定子密钥的工作量。通过全局变量可激活附加的安全机制，以便安全级别能适应于各个子密钥。BACKUP 为 1 表示将其他后备参数均置为 1。

使用后备选项进行多重加密时，与单重加密使用的主密钥和二级密钥相同。加密的轮数放在每次加密前的中间密文上，中间密文存储在附加加密的输出数组中。

## 1.38.3 子密钥

所有的子密钥都将明文存放在中间变量，这些变量都是 64 比特的无符号整型字。通常分组的长度会产生剩余部分，需进行适当调整，使剩余部分无论何时进行运算，其结果都被屏蔽为剩余部分的长度。屏蔽变量 LMASK 用于控制所进行的操作，该变量是经过适当调整的、由 1 组成的块。屏蔽码是一个 64 比特的字，当分组的长度是 64 的整整数倍时，屏蔽码为一个全 1 的字；当分组的长度为 $a \bmod 64$ 时，则屏蔽码的低 $a$ 比特为 1。例如，当分组长度为 11 模 64 时，屏蔽码为 0x7FF。

每个子密钥都是通过反复混乱的状态变量，将密钥和二级密钥 SPICE 混乱进去的。每个混乱步骤都是可逆的，脱密就是其逆过程。在大多数情况下，相应的逆变换是显而易见的。

每个子密钥在开始时将一些扩展密钥与明文相加，并通过相加更多的扩展密钥来产生最终的输出。密钥字的选取与分组长度有关，如果分组长度是 $B$ 比特，就将 KX 表中数组的第 $B$ 个字与明文的第 0 个字相加来初始化状态变量 $s_0$；KX[$B$+1]与明文的第 1 个字相加来初始化状态变量 $s_1$，依次类推，然后运行特定的子密钥算法。在最后输出前，数组 KX[$B$+8]加到 $s_0$，生成密文字 0 的输出，KX[$B$+9]加到 $s_1$，生成密文字 1 的输出，依次类推。

### 1．HPC-Medium，分组长度为 65～128 比特

明文存放在变量 $s_0$ 和 $s_1$ 中，$s_1$ 包含适当调整的零碎部分。KX 表中数组的两个连续字与 $s_0$ 和 $s_1$ 相加，KX[分组长度]也加到 $s_0$。混乱由 8 轮组成，记为 0～7，下面伪代码中变量 $I$ 表示轮数。轮算法如下：

```
k = KX[s0&255]
s1 += k
s0 ^= k<<8
s1 ^= s0
```

```
s1 &= lmask Fragment masking example. Other uses omitted.
s0 -= s1>>11
s0 ^= s1<<2
s0 -= spice[i^4]
s0 += (s0<<32) ^ (PI19 + blockSize)
s0 ^= s0>>17;   s0 ^= s0>>34
t = spice[i]
s0 ^= t
s0 += t<<5
t >>= 4
s1 += t
s0 ^= t
s0 += s0 << (22 + (s0&31))
s0 ^= s0>>23
s0 -= spice[i^7]
t = s0&255
k = KX[t]
kk = KX[t+3*i+1]
s1 ^= k
s0 ^= kk<<8
kk ^= k
s1 += kk>>5
s0 -= kk<<12
s0 ^= kk &～ 255
s1 += s0
s0 += s1<<3
s0 ^= spice[i^2]
s0 += KX[blockSize+16+i]
s0 += s0<<22
s0 ^= s1>>4
s0 += spice[i^1]
s0 ^= s0>>(33+i)
```

8 轮完成后，KX[blockSize+8]与 $s_0$ 相加，KX 的下一个字与 $s_1$ 相加。$s_0$ 和 $s_1$ 存放在输出数组中，$s_1$ 被屏蔽，其任何高比特（屏蔽的左边）不发生改变。

脱密可通过逆向加密步骤进行，次序相反，各步由其反函数代替。轮数从 7 到 0，如 $s_0$+=$s_1$ 变为 $s_0$–=$s_1$。

### 2．HPC-Long，分组长度为 129～512 比特

该子密钥有 8 个状态变量，即 $s_0$,…,$s_7$。将明文存放到 $s_0$,…,$s_7$ 中，$s_0$ 取明文的第 0 个字，$s_1$ 取下一个字，依次类推，零碎部分总放在 $s_7$ 中，其他变量按顺序使用。$s_0$ 和 $s_1$ 总要被用到，

$s_2$只在分组的长度超过 192 比特时才使用，依次类推。对该子密钥的最小分组长度为 129 比特，只使用 $s_0$、$s_1$和 $s_7$的低比特；如果最大分组长度为 512 比特，则使用全部 8 个状态变量的所有比特。

KX 表中数组中的 8 个连续字与 $s_0$,$s_1$,…,$s_7$相加，KX[(分组长度&255)+0]与 $s_0$相加，KX 表中的下一字与 $s_1$相加，依次类推。但无论块的大小，KX[(分组长度&255)+7]总与 $s_7$相加。

HPC-Long 的混乱函数与 HPC-Medium 的类似，共有 8 轮，$i$ 是轮数，取值为 0～7，如下伪代码所示。

```
t = s0&255
k = KX[t]
kk = KX[t+3*i+1]
kk s1 += k
s0 ^= kk<<8
s1 += kk>>5
s0 -= kk<<12
s7 += kk
s7 ^= s0
s1 += s7
s1 ^= s7<<13
s0 -= s7>>11
s0 += spice[i]
s1 ^= spice[i^1]
s0 += s1<<(9+i)
s1 += (s0>>3) ^ (PI19+blockSize)
s0 ^= s1>>4
s0 += spice[i^2]
t = spice[i^4]
s1 += t
s1 ^= t>>3
s1 -= t<<5
s0 ^= s1
```

混乱函数的执行码与分组长度有关，如表 1.38.2 所示。

表 1.38.2　混乱函数的执行码与分组长度的关系

| 分组长度 | 执行码 |
|---|---|
| 448 | $s_6$ += $s_0$; $s_6$ ^= $s_3$<<11; $s_1$ += $s_6$>>13; $s_6$ += $s_5$<<7; $s_4$ ^= $s_6$; |
| 384 | $s_5$ ^= $s_1$; $s_5$ += $s_4$<<15; $s_0$ -= $s_5$>>7;　$s_5$ ^= $s_3$>>9; $s_2$ ^= $s_5$; |
| 320 | $s_4$ -= $s_2$; $s_4$ ^= $s_1$>>10; $s_0$ ^= $s_4$<<3;　$s_4$ -= $s_2$<<6; $s_3$ += $s_4$; |
| 256 | $s_3$ ^= $s_2$; $s_3$ -= $s_0$>>7;　$s_2$ ^= $s_3$<<15; $s_3$ ^= $s_1$<<5; $s_1$ += $s_3$; |
| 192 | $s_2$ ^= $s_1$; $s_2$ += $s_0$<<13; $s_1$ -= $s_2$>>5;　$s_2$ -= $s_1$>>8; $s_0$ ^= $s_2$; |

对较大的分组长度，也执行较小分组长度的执行码。

```
s1 ^= KX[(blockSize + 17 + (i<<5))&255]
s1 += s0<<19
s0 -= s1>>27
s1 ^= SPICE[i^7]
s7 -= s1
s0 += s1 & s1>>5
s1 ^= s0 >> (s0 & 31)
s0 ^= KX[s1&255]
```

完成 8 轮后，KX[分组长度+8]与 $s_0$ 相加，接下来的 7 个 KX 表中的字分别与 $s_1,\cdots,s_7$ 相加；KX[分组长度+15]总是与 $s_7$ 相加，而不管 $s_1,\cdots,s_6$ 中任何未使用的变量，最后将它们存放在输出中。

脱密是通过逆向加密步骤进行的，轮数为 7～0。

### 3．HPC-Extended，分组长度为 513 比特或更长

该子密钥的状态变量仍是 $s_0,\cdots,s_7$。由于此时分组长度超过状态变量的总长度，需要用新办法：前 7 个状态变量按通常的方法使用，$s_7$ 是一个“交换的”状态变量，绝大多数明文被交换（一次一字）混乱进 $s_7$ 中，然后又交换出来。明文用该方法（交换进去、混乱、交换出来）处理三次。

开始加密前，根据块长计算以下数据：

LWD 是输入的字数，即块长/64；LMASK 是经适当调整的、用于剩余部分的 1 的屏蔽变量；QMSK 是一个大于或等于 LWD 的 2 的最小幂；SWZ 是一个超过 QMSK 的最小的 Swizpoly 数；Swizpoly 数即 0、3、7、0xb、0x13、0x25、0x43、0x83、0x11d、0x211、0x409、0x805、0x1053、0x201b、0x402b、0x8003、0x1002d、0x20009、0x40027、0x80027、0x100009、0x200005、0x400003、0x800021、0x100001b、0x2000009、0x4000047、0x8000027、0x10000009、0x20000005、0x40000053、0x80000009。

对于 513 比特的块，LWD=9、LMASK=1、QMSK=15、SWZ=0x13=19；对于 1024 比特的块，LWD=16、LMASK=0xffffffffffffffff（全为 1），QMSK 和 SWZ 不变。如果将块增加到 1025 比特，则 LWD=17，LMASK 变回到 1，QMSK 跳到 31，SWZ 变为 0x25=37。

初始化向量的准备：$s_0,\cdots,s_7$ 存放明文的前 8 个字，并与 KX 表中数组相加，即

$$s_0=\text{ptx}[0]+\text{KX}[\text{blockSize}\&255]$$
$$s_1=\text{ptx}[1]+\text{KX}[\text{blockSize}\&255+1]$$
$$\cdots$$
$$s_7=\text{ptx}[7]+\text{KX}[\text{blockSize}\&255+7]$$

现在开始混乱操作。混乱函数通过各种附加输入对 $s_0,\cdots,s_7$ 进行操作，包括轮索引 $i$。LMASK 与包含来自 ptx[7]的 64 比特明文的 $s_7$ 一致。混乱前的过程是：混乱函数调用 4 次（$i$=0,…,3），$s_7$ 写到密文数组的第 7 个字中，用于交换；然后 $i$ 清 0。从明文第 8 个字，即 ptx[8]开始，将明文的每个字送入 $s_7$，不与 KX 表中数组的任何字相加。设置 LMASK 与 $s_7$ 中的有效比特匹配，直到明文的最后一个字被送入 $s_7$，再根据剩余部分的大小重新计算，并将 $s_7$ 屏蔽，混乱函数执行一次。$s_7$ 写入密文（输出）数组中与交换进明文字一样的相关位置，这样对明文的第一步的处理就结束了。除了第 0～6 个字保存在 $s_0,\cdots,s_6$ 中，其余已处理的数据保存在输出块中。

第二步的数据准备工作是，处理了最后一个明文字并输出后，$s_7$ 从密文的第 7 个字，即 ctx[7]开始重新装载，LMASK 置为全 1，再次运行混乱函数。块长加到 $s_0$，混乱函数调用三次（轮索引为 0～2）。块长再加到 $s_0$，轮索引重置为 0，$s_7$ 写入 ctx[7]。第二步的数据处理是将数据字每次一个地交换进 $s_7$，但有两点不同：首先，这里的数据是从临时存放的密文数组中写入的；其次，对该数据字的处理顺序不同（相关要求在后文中叙述）。从 ctx[8]到 ctx[LWD–1]的每一个密文字都交换进 $s_7$ 中并处理一次，并回写到 ctx 数组的相同位置。对每个字的处理就是运行一次混乱函数。

第三步的数据准备工作是，$s_7$ 从 ctx[7]重新装载，LMASK 置为全 1，轮索引置为 1，并运行混乱函数。块长加到 $s_0$，混乱函数运行三次（轮索引为 0～2）。块长再加到 $s_0$，$s_7$ 再存入 ctx[7]，轮索引置为 0。第三步的数据处理过程与第二步过程一样，每次一个地将来自 ctx 数组的数据字交换入 $s_7$，设置 LMASK，混乱函数执行一次，将 $s_7$ 写回 ctx 数组相同位置，该步使用与前两步不同的选取顺序（在后文中叙述）。

最后的准备工作是将 $s_7$ 从 ctx[7]重新装载，LMASK 置为全 1。轮索引置为 0，运行混乱函数一次，然后运行三次混乱函数（索引值为 0～2），最后将 KX 数组的 8 个字分别与 $s_0,\cdots,s_7$ 相加，结果存入 ctx[0],…,ctx[7]。

ctx[0]=$s_0$+KX[(blockSize&255)+8]，　blockSize 是块长

ctx[1]=$s_1$+KX[(blockSize&255)+9]

…

ctx[7]=$s_7$+KX[(blockSize&255)+15]

该子密钥的混乱函数与 HPC-Long 的类似，$i$ 是轮索引，每次使用前和交换进/出时，必须屏蔽 $s_7$。

```
t = s0 & 255
k = KX[t]
kk = KX[t+1+(i<<2)]
s3 += s7
```

```
s5 ^= s7
s1 += k
s2 ^= k
s4 += kk
s6 ^= kk
s4 ^= s1
s5 += s2
s0 ^= s5>>13
s1 -= s6>>22
s2 ^= s7<<7
s7 ^= s6<<9
s7 += s0
t = s1 & 31
tt = s1>>t
ttt = s2<<t
s3 += ttt
s4 -= s0
s5 ^= ttt
s6 ^= tt
s7 += tt
t = s4>>t
s2 -= t
s5 += t
```

如果轮索引 $i$ 为 1，SPICE 就同 $s_0,\cdots,s_7$ 结合，即

```
s0 += SPICE[0]; s1 ^= SPICE[1]; s2 -= SPICE[2]; s3 ^=SPICE[3]
s4 += SPICE[4]; s5 ^= SPICE[5]; s6 -= SPICE[6]; s7 ^=SPICE[7]
s7 -= s3
s1 ^= s7>>11
s6 += s3
s0 ^= s6
```

下列伪代码用于交换 $s_2$ 和 $s_5$ 中的偶数比特位，$s_3$ 和 $s_0$ 是被修改的对象。

```
t = s2^s5
s3 -= t
t &= 0x5555555555555555
s2 ^= t;  s5 ^= t
s0 += t
t = s4<<9
s6 -= t
s1 += t
```

这里定义了第二步和第三步的数据选取顺序，它们是两种不同的迭代方法，每一步以伪随机顺序遍历 $N$=log2(QMSK+1)比特的所有值。

第一种迭代方法用于第二步：

$$Qnew=5Q+1$$

$$Qnew \&= QMSK$$

这以 $Q$=0 开始和结束。每步之后都以 QMSK 来屏蔽 $Q$，将 $Q$ 限制为一个 $N$ 比特的值。当 $Q$ 的取值小于 8 或大于 LWD 时，就重做该步。当 $Q$ 的取值为[8，LWD]时，字 ctx[$Q$]交换进 $s_7$，混乱后再交换出击。

第二种迭代方法用于第三步：

QMSK 增加变成 2 的一个的幂。

```
Qnew <<= 1
if (Qnew & QMSK)
them Qnew ^= SWZ
```

迭代以 $Q$=1 开始和结束。当 $Q$ 的取值为[8，LWD]时，就将 ctx[$Q$]换进 $s_7$ 进行混乱后再交换出击。SWZ 的值可从 Swizpoly 数中取得，通常取其为大于 2$N$ 的最小 Swizpoly 数。

#### 4．HPC-Short，分组长度为 36～64 比特

将明文存放在变量 $s_0$ 的右边，使用 LMASK 为 1 的块来屏蔽 $s_0$，使 $s_0$ 的有效比特参加运算，将字 KX[分组块长]加到 $s_0$ 上，计算如下数据：

$$LBH = (blockSize+1)/2，\quad blockSize 是块长$$

$$LBQ = (LBH+1)/2$$

$$LBT = (blockSize+LBQ)/4 + 2$$

$$GAP = 64 – blockSize$$

然后混乱 8 次（$i$ 取 0 到 7），再将 KX[blockSize+8]加到 $s_0$ 上。屏蔽 $s_0$ 后，将有效比特写到输出数组中。

混乱函数的伪代码为：

```
k = KX[s0&255] + SPICE[i]
s0 += k<<8
s0 ^= (k>>GAP) &～255
s0 += s0<<(LBH+i)
t = SPICE[i^7]
s0 ^= t
s0 -= t>>(GAP+i)
s0 += t>>13
```

```
s0 ^= s0>>LBH
t = s0&255
k = KX[t]
k ^= SPICE[i^4]
k = KX[t+3*i+1] + (k>>23) + (k<<41)
s0 ^= k<<8
s0 -= (k>>GAP) &～255
s0 -= s0<<LBH
t = SPICE[i^1] ^ (PI19+blockSize)
s0 += t<<3
s0 ^= t>>(GAP+2)
s0 -= t
s0 ^= s0>>LBQ
s0 += Permb[s0&15]
t = spice[i^2]
s0 ^= t>>(GAP+4)
s0 += s0<<(LBT + (s0&15))
s0 += t
s0 ^= s0>>LBH
Permb[16]:
0xB7E151628AED2A6A -0,   0xBF7158809CF4F3C7 -1,
0x62E7160F38B4DA56 -2,   0xA784D9045190CFEF -3,
0x324E7738926CFBE5 -4,   0xF4BF8D8D8C31D763 -5,
0xDA06C80ABB1185EB -6,   0x4F7C7B5757F59584 -7,
0x90CFD47D7C19BB42 -8,   0x158D9554F7B46BCE -9,
0x8A9A276BCFBFA1C8 -10,  0xE5AB6ADD835FD1A0 -11,
0x86D1BF275B9B241D -12,  0xF0D3D37BE67008E1 -13,
0x0FF8EC6D31BEB5CC -14,  0xEB64749A47DFDFB9 -15.
```

脱密的过程是逆向进行的，需要注意的是，$s_0$+=Permb[$s_0$&15]要由 $s_0$–=Permbi[$s_0$&15]代替。

```
Permbi[16]:
0xE5AB6ADD835FD1A0 -11,  0xF0D3D37BE67008E1 -13,
0x90CFD47D7C19BB42 -8,   0xF4BF8D8D8C31D763 -5,
0x4F7C7B5757F59584 -7,   0x324E7738926CFBE5 -4,
0x62E7160F38B4DA56 -2,   0xBF7158809CF4F3C7 -1,
0x8A9A276BCFBFA1C8 -10,  0xEB64749A47DFDFB9 -15,
0xB7E151628AED2A6A -0,   0xDA06C80ABB1185EB -6,
0x0FF8EC6D31BEB5CC -14,  0x86D1BF275B9B241D -12,
0x158D9554F7B46BCE -9,   0xA784D9045190CFEF -3.
```

### 5．HPC-Tiny，分组长度为 0～35 比特

分组长度 0 的加密是空操作，立即返回。

对于不同的分组长度，HPC-Tiny 可细分为多个子子密钥。一个密钥扩展数组对所有子子密钥都有用，为了包括所有的二级密钥和足够的 KX 表中数组，根据输入的密钥和二级密钥，递归调用一次 HPC-Medium 或 HPC-Long 生成一个伪随机数，由这个伪随机数控制明文的置换。

所有的子子密钥都和其他子密钥一样，将明文放置于变量 $s_0$ 右边的、LMASK 置为 1 的块，用于以屏蔽 $s_0$，使其有效比特参加运算。

对于分组长度为 1～4 比特的子子密钥，在开始时调用一次 HPC-Medium，将 KX[16+2×blockSize]和 KX[17+2×blockSize]复制进数组 tmp[]中。使用相同的 KX 表中数组和 SPICE，以 128 比特的块来调用 HPC-Medium 来加密，其结果作为两个字的伪随机数来控制在 $s_0$ 上进行的置换。

如果分组长度为 1 比特，则提取伪随机数的单个比特，将其与 $s_0$ 进行模 2 加。将 tmp[0]与 tmp[1]相加后作为低 64 比特，tmp[1]作为高 64 比特，因此可得到一个 128 比特的数 $N$，再调用 Fibonacci 压缩，将 $N$ 缩减为 1 个比特。

$$N \mathrel{+}= N>>89 \qquad N \mathrel{\hat{}}= N>>55$$
$$N \mathrel{+}= N>>34 \qquad N \mathrel{\hat{}}= N>>21$$
$$N \mathrel{+}= N>>13 \qquad N \mathrel{\hat{}}= N>>8$$
$$N \mathrel{+}= N>>5 \qquad N \mathrel{\hat{}}= N>>3$$
$$N \mathrel{+}= N>>2 \qquad N \mathrel{\hat{}}= N>>1$$
$$N \mathrel{+}= N>>1$$

然后将 $N$ 的最低有效比特异或到 $s_0$ 中。注意：脱密和加密的方法是一样的。

如果分组长度为 2 和 3 比特，则 tmp 数组的每个字都用于控制在 $s_0$ 上进行的置换。先使用 tmp[0]，低比特异或到 $s_0$ 上，高比特加到 $s_0$ 上，$s_0$ 再循环左移 1 位，并使临时变量使用的 2 倍分组长度比特被右移掉。为了完全利用每一个变量，要执行足够多的圈数（16 圈或 11 圈）。脱密是加密的逆过程。

如果分组长度为 4 比特，同分组长度为 2 和 3 比特的情形一样，tmp 数组的每一个字都用于控制置换。每个临时变量都从低位开始，按 4 比特来使用。要用完每一个变量，需要 8 圈，每圈都将 4 比特的块加到 $s_0$ 中，然后使用 PERM1 置换；再将下一个 4 比特的块异或到 $s_0$，并使用 PERM2 置换。脱密是加密的逆过程。

```
PERM1:   0x324f6a850d19e7cb   /* cycle notation (0B6D49851CF3E27)(A) */
PERM1I:  0xc3610a492b8dfe57   /* inverse of PERM1 */
PERM2:   0x2b7e1568adf09c43   /* cycles (0396D7A5F2CEB14)(8) */
PERM2I:  0x5c62e738d9a10fb4   /* inverse of PERM2 */
```

对于 PERM1 置换，应是 0→b,1→c,…,15→3。

分组长度为 5 和 6 比特的情形与 4 比特的情形相似，可使用一个更长的 tmp 数组，以提供更多的伪随机数。这使 32 或 64 个值的置换成为可能。对于分组长度为 5 比特的情形，使用 KX 表中数组的 3 个字（192 比特）；对于分组长度为 6 比特的情形，使用 6 个字（384 比特）。两种情形的开始字都是 KX[16+2×分组块长]，使用 KX 表中数组作为密钥扩展数组，调用 HPC-Long 加密这些 KX 表中的数组的字，并使用调用 HPC-Tiny 时所用的 SPICE。加密的结果放在临时数组中，并使用其每个值（从 tmp[0]开始）控制在 $s_0$ 上进行的置换。对于分组长度为比特 5 的情形，要循环 7 次，每次循环后，控制变量 $T$ 右移 9 比特。循环伪代码如下：

```
s0 ^= T
s0的低 4 比特映射到 PERM1
s0 ^= s0 >>3
s0 += T>>blockSize
s0低 4 比特映射到 PERM2
```

对于分组长度为 6 比特的情形，循环 6 次，$T$ 右移 11 比特。

对于分组长度为 7～16 比特的情形，使用与分组长度为 16～35 比特相同的方法生成 10 个字的伪随机数组 tmp，置 $T$=tmp[0]。每次循环都使用 $T$ 的低 2×分组块长比特。循环伪代码如下：

```
s0 += T
s0 ^= KX [16*I + (s0&15)] <<4
```

注：$I$ 是 tmp 数组的索引号，为 0～9。

```
s0右移 4 比特
s0 ^= s0 >> LBH
s0 ^= T >> blockSize  (blockSize是块长)
s0 += s0 << (LBH+2)
s0 ^= Perma[s0&15]
s0 += s0 << LBH
LBH is (blockSize+1)/2,四舍五入
```

其中：

```
Perma[16]:
0x243F6A8885A308D3 ^0,   0x13198A2E03707344 ^1,
0xA4093822299F31D0 ^2,   0x082EFA98EC4E6C89 ^3,
0x452821E638D01377 ^4,   0xBE5466CF34E90C6C ^5,
0xC0AC29B7C97C50DD ^6,   0x9216D5D98979FB1B ^7,
0xB8E1AFED6A267E96 ^8,   0xA458FEA3F4933D7E ^9,
0x0D95748F728EB658 ^10,  0x7B54A41DC25A59B5 ^11,
```

```
0xCA417918B8DB38EF ^12,  0xB3EE1411636FBC2A ^13,
0x61D809CCFB21A991 ^14,  0x487CAC605DEC8032 ^15.
```

脱密是通过逆向执行加密过程来实现的，$s_0$ ^= Perma[$s_0$&15] 的逆是 $s_0$ ^= Permai[$s_0$&15]。

其中：

```
Permai[16]:
0xA4093822299F31D0 ^2,   0x61D809CCFB21A991 ^14,
0x487CAC605DEC8032 ^15,  0x243F6A8885A308D3 ^0,
0x13198A2E03707344 ^1,   0x7B54A41DC25A59B5 ^11,
0xB8E1AFED6A267E96 ^8,   0x452821E638D01377 ^4,
0x0D95748F728EB658 ^10,  0x082EFA98EC4E6C89 ^3,
0xB3EE1411636FBC2A ^13,  0x9216D5D98979FB1B ^7,
0xBE5466CF34E90C6C ^5,   0xC0AC29B7C97C50DD ^6,
0xA458FEA3F4933D7E ^9,   0xCA417918B8DB38EF ^12.
```

对于分组长度为 16～35 比特的情形，首先使用与分组长度为 7～15 比特相同的方法，产生 10 个字的临时数组 tmp，前 8 个字是密钥 SPICE 与 KX 表中数组字的异或，即

```
tmp[0] = spice[0] ^ KX[4*blockSize+16]
…
tmp[7] = spice[7] ^ KX[4*blockSize+23]
```

然后以 KX 为密钥，SPICE 为全 0，调用 HPC-Long 加密 512 比特的 tmp[0]…tmp[7]。tmp[8]可通过以下伪代码迭代来产生：

```
tmp[8] += ((tmp[8]<<21) + (tmp[8]>>13)) ^ (tmp[i] + KX[16+i])
```

这里 $i$ 的为 0～7，tmp[9]是 tmp[8]的异或。

以较短分组长度类似的情形处理 10 个字的临时数组 tmp，其循环伪代码如下：

```
s0 += t
s0 ^= KX[s0&255] << 8
s0 &= LMASK
s0 右移 8 比特
t >>= blockSize
```

这里 $t$ 是 tmp 数组的字。

如果分组长度为 35～32 比特，则循环 2 圈；如果分组长度为 32～22 比特，则循环 3 圈；如果分组长度 22～16 比特，则循环 4 圈。

HPC 加密算法的一个突出优点是可以加密任意长度的分组块，且不需要进行数据扩展。

## 1.39　MAGENTA 加密算法

MAGENTA（通用加密和网络电信的多功能算法）加密算法是由德国 Deutsche 电信公司的 K. Huber 提交的，其设计者是 M. J. Jacobson 和 K. Huber。MAGENTA 采用了 Feistel 结构（即 DES 型），其核心部分以快速哈特莱变换（Fast Hartly Transform，FHT）为基础。

### 1．定义符号和函数

假设，$B$ 为 8 比特二进制数集，即 $B=\{(x_7,x_6,\cdots,x_0)|x_i\in\{0,1\}\}$。对于任意 $x\in B$，设 $x=(x_7,\cdots,x_0)$，可将 $x$ 与{0,1,⋯,255}中的一个整数相对应，即$(x_7,\cdots,x_0)\to x_7 2^7+x_6 2^6+\cdots+x_0$，$B$ 中元素就可用{0,1,⋯,255}中的整数来表示。

定义 GF($2^8$)是极小多项式为 $p(\lambda)=\lambda^8+\lambda^6+\lambda^5+\lambda^2+1$ 的有限域，$\alpha$ 是 GF($2^8$)的本原元素，$p(\alpha)=0$。

① $f(x)$：$B\to B$。对一切 $x\in B$，定义：

$$f(x)=\begin{cases}\alpha^x, & x\neq 255\\ 0, & x=255\end{cases}$$

② $A(x,y)$：$B^2\to B$。对一切$(x,y)\in B^2$，定义：

$$A(x,y)=f[x\oplus f(y)]$$

③ PE$(x,y)$：$B^2\to B^2$。对一切$(x,y)\in B^2$，定义：

$$\mathrm{PE}(x,y)=[A(x,y),A(y,x)]=\{f[x\oplus f(y)],f[y\oplus f(x)]\}$$

④ $\pi(x_0,x_1,\cdots,x_{15})$：$B^{16}\to B^{16}$。对一切$(x_0,x_1,\cdots,x_{15})\in B^{16}$，其中 $x_i(i=0,\cdots,15)\in B$，定义：

$$\pi(x_0,x_1,\cdots,x_{15})=[\mathrm{PE}(x_0,x_8),\mathrm{PE}(x_1,x_9),\cdots,\mathrm{PE}(x_7,x_{15})]$$

⑤ $T(x_0,x_1,\cdots,x_{15})$：$B^{16}\to B^{16}$，即$\{0,1\}^{128}\to\{0,1\}^{128}$。对于任意$(x_0,x_1,\cdots,x_{15})\in B^{16}$，定义：

$$T(x_0,x_1,\cdots,x_{15})=\pi\{\pi\{\pi[\pi(x_0,x_1,\cdots,x_{15})]\}\}$$

⑥ $C^{(i)}(x_0,x_1,\cdots,x_{15})$：$B^{16}\to B^{16}$。设 $X=(x_0,x_1,\cdots,x_{15})\in B^{16}$，记 $X_\mathrm{e}=(x_0,x_2,\cdots,x_{14})$，即取 8 个偶数字节；$X_\mathrm{o}=(x_1,x_3,\cdots,x_{15})$，即取 8 个奇数字节。对于任意$(x_0,x_1,\cdots,x_{15})\in B^{16}$，递归地定义：

$$C^{(j+1)}(x_0,\cdots,x_{15})=T[(x_0,\cdots,x_7)\oplus C_\mathrm{e}^{(j)},(x_8,\cdots,x_{15})\oplus C_\mathrm{o}^{(j)}],\qquad j\geqslant 1$$

式中，$C^{(1)}=T(x_0,\cdots,x_{15})$，$C_\mathrm{e}^{(j)}$为取第 $j$ 次迭代结果 $C^{(j)}$的 8 个偶数字节，$C_\mathrm{o}^{(j)}$为取第 $j$ 次迭代结果 $C^{(j)}$的 8 个奇数字节。

⑦ $E^{(r)}$：$B^{16}\to B^8$，对于某个 $r$，定义：

$$E^{(r)}(x_0,\cdots,x_{15})=C_e^{(r)}$$

式中，$C_e^{(r)}$为第 $r$ 次迭代结果 $C^{(r)}$的 8 个偶数字节。

### 2．MAGENTA 加/脱密算法

MAGENTA 采用 Feistel 结构，取 $E^{(3)}$为核心变化函数，其 Feistel 圈可表示成：

$$F_Y^{(X)}=[(x_8,\cdots,x_{15}),(x_0,\cdots,x_7)\oplus E^{(3)},(x_8,\cdots,x_{15},x_0,\cdots,x_7)]$$

设明文 $M=(x_0,\cdots,x_{15})\in B^{16}$，MAGENTA 加/脱密算法的密钥可以有三种情形：

对 128 比特，密钥为：　　$K=(K_1,K_2)$

对 192 比特，密钥为：　　$K=(K_1,K_2,K_3)$

对 256 比特，密钥为：　　$K=(K_1,K_2,K_3,K_4)$

式中，$K_1=(Y_0,\cdots,Y_7)$，$K_2=(Y_8,\cdots,Y_{15})$，$K_3=(Y_{16},\cdots,Y_{23})$，$K_4=(Y_{24},\cdots,Y_{31})$。

MAGENTA 加/脱密算法分别记为 EMA 和 DMA。设密文分组为 $C$，有：

$$C=\mathrm{EMA}_K^{(M)}=\begin{cases}F_{K1}(F_{K1}(F_{K2}(F_{K2}(F_{K1}(F_{K1}(M)))))), & K=(K_1,K_2)\in B^{16}\\ F_{K1}(F_{K2}(F_{K3}(F_{K3}(F_{K2}(F_{K1}(M)))))), & K=(K_1,K_2,K_3)\in B^{24}\\ F_{K1}(F_{K2}F_{K3}(F_{K4}(F_{K4}(F_{K3}(F_{K2}(F_{K1}(M))))))), & K=(K_1,K_2,K_3,K_4)\in B^{32}\end{cases}$$

从上式可看出，当密钥长度为 128 或 192 比特时，MAGENTA 加密算法要执行 6 个 Feistel 圈。当密钥长度为 256 比特时，则执行 8 个 Feistel 圈。定义 $V(x_0,\cdots,x_{15})=(x_8,\cdots,x_{15},x_0,\cdots,x_7)$，即将高 8 字节与低 8 字节进行交换，其脱密算法为：

$$\begin{aligned}M&=\mathrm{DMA}_K(C)=V(\mathrm{EMA}_K(V(C)))\\&=\begin{cases}V(F_{K1}(F_{K1}(F_{K2}(F_{K2}(F_{K1}(F_{K1}(V(C))))))), & K=(K_1,K_2)\in B^{16}\\ V(F_{K1}(F_{K2}(F_{K3}(F_{K3}(F_{K2}(F_{K1}(V(C))))))), & K=(K_1,K_2,K_3)\in B^{24}\\ V(F_{K1}(F_{K2}F_{K3}(F_{K4}(F_{K4}(F_{K3}(F_{K2}(F_{K1}(V(C))))))), & K=(K_1,K_2,K_3,K_4)\in B^{32}\end{cases}\end{aligned}$$

从上面加/脱密算法可见，MAGENTA 加密算法与脱密算法可以说是完全一样的。脱密算法完全可用加密算法来实现，只是在脱密时先将输入密文分组的高 8 字节与低 8 字节交换一下，然后交换经加密算法输出的高、低两部分即可得到明文分组。

## 1.40　MARS 加密算法

MARS 加密算法是美国 IBM 公司于 1998 年 6 月提交的，其设计者是 Carolynn、Burwick 等。MARS 加密算法的明文分组为 128 比特，密钥长度在 128～1248 比特之间可变，这样

可增强适用性，若 Diffie-Hellman 协议交换的密钥远大于 256 比特，也可以用于 MARS 加密算法。

### 1．加密算法框架

MARS 加密算法的输入和输出均为 4 个 32 比特的字，从结构框架来看可分成三个部分，如图 1.40.1。

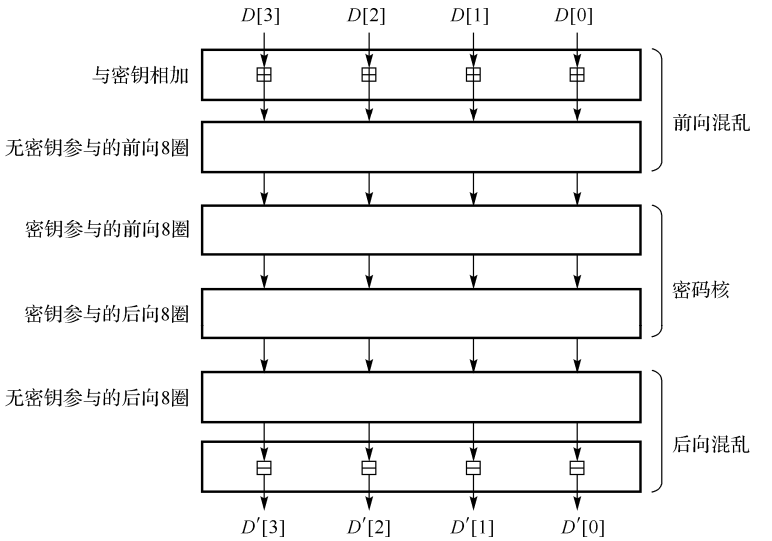



图 1.40.1　MARS 加密算法结构框架

第一部分是前向混乱，先将数据字与密钥相加，接着是无密钥参与的 8 圈（Feistel 类型）变换。

第二部分是密码核，是密钥参与的 16 圈（Feistel 类型）变换，可分为密钥参与的前向 8 圈和密钥参与的后向 8 圈。

第三部分是后向混乱，经第二部分处理后的数据再经无密钥参与的后向 8 圈（Feistel 类型）变换，再减去密钥字，即可得到密文。

符号约定：

$D[\,]$：4 个 32 比特的字明文，加密处理后的密文为 $D'[\,]$。

$K[\,]$：扩展的密钥序列，由 40 个 32 比特的字组成。

$S[\,]$：$S$ 盒由 512 个 32 比特的字组成，前 256 个字为 $S_0$，后 256 个字为 $S_1$。

### 2．各部分的结构

（1）前向混乱部分的结构如图 1.40.2 所示。

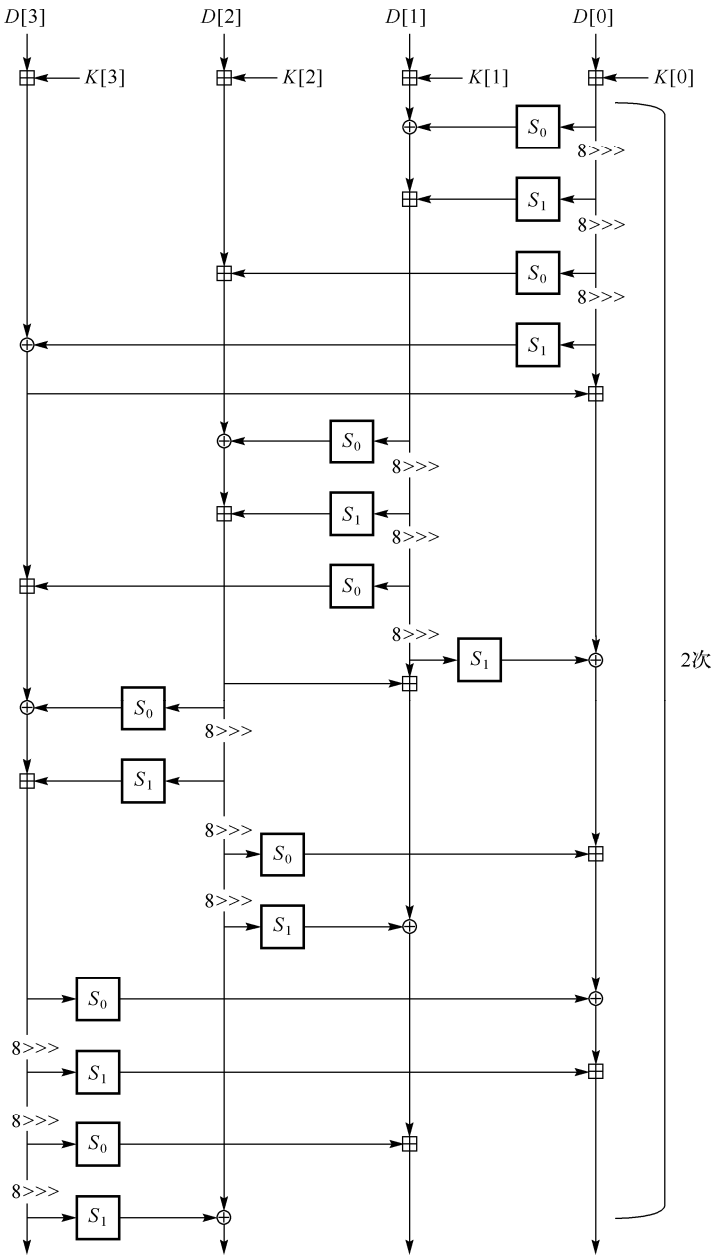



图 1.40.2　前向混乱部分的结构

首先将密钥字 $K[0]$、$K[1]$、$K[2]$、$K[3]$分别加到 $D[0]$、$D[1]$、$D[2]$、$D[3]$上，这里的“⊞”表示模 $2^{32}$ 加。即

$$D[i]=D[i]\boxplus K[i],\qquad i=0,1,2,3$$

其次进行前向混乱 8 圈。对 $i$=0～7，进行如下变换。

$D[1]=D[1]\oplus S_0[D[0]$的最低字节$]$（以[ ]中的数 0～255 作为地址取 $S0$ 中的元素）

$D[1]=D[1]\boxplus S_1[D[0]$的第二字节$]$（以[ ]中的数 0～255 作为地址取 $S1$ 中的元素）

$$D[2]=D[2]\boxplus S_0[D[0]\text{的第三字节}]$$

$$D[3]=D[3]\oplus S_1[D[0]\text{的高字节}]$$

$D[0]=D[0]>>24$（$D[0]$循环右移 24 位，即图 1.40.2 中循环右移 3 次，每次 8 位）

如果 $i$=0 或 4，则

$$D[0]=D[0]\boxplus D[3]$$

如果 $i$=1 或 5，则

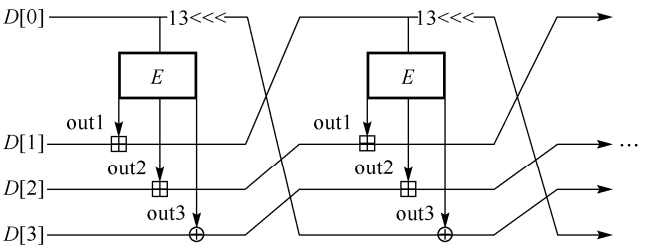
$$D[0]=D[0]\boxplus D[1]$$

最后将 $D$[ ]循环右移 1 个字，即

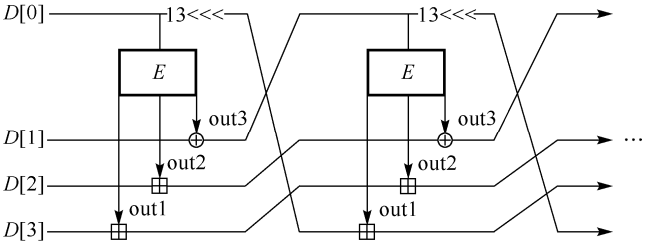
$$(D[3],D[2],D[1],D[0])\leftarrow(D[0],D[3],D[2],D[1])$$

至此就完成了一圈前向混乱。

（2）密码核部分。本部分由 16 圈（Feistel 类型）组成，每圈使用 1 个密钥参与的 $E$ 函数，如图 1.40.3 所示。




（a）前向模式




（b）后向模式

图 1.40.3　前向模式和后向模式

*E* 函数以 1 个数据字和 2 个密钥字为输入，其输出为 3 个字，设 *E* 函数输出的 3 个字分别为 out1、out2、out3。16 圈变换如下：

对于 $i$=0～5，进行如下变换：

$$(\text{out1},\text{out2},\text{out3})=E(D[0],K[2i+4],K[2i+5])$$

$$D[0]=D[0]\lll 13\text{（}D[0]\text{循环左移 13 位）}$$

$$D[2]=D[2]\boxplus \text{out2}$$

如果 $i$<8，则前 8 圈进行前向混乱（前向模式）：

$$D[1]=D[1]\boxplus \text{out1}$$

$$D[3]=D[3]\oplus \text{out3}$$

否则后 8 圈进行后向混乱（后向模式）：

$$D[1]=D[1]\oplus \text{out3}$$

$$D[3]=D[3]\boxplus \text{out1}$$

$$(D[3],D[2],D[1],D[0])\leftarrow(D[0],D[3],D[2],D[1])\text{（}D[\ ]\text{循环右移 1 个字）}$$

*E* 函数的逻辑框图如图 1.40.4 所示。

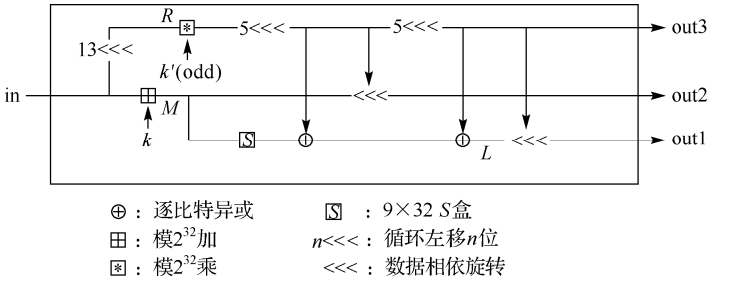



图 1.40.4　*E* 函数逻辑框图

*E* 函数的输入是 1 个数据字和 2 个密钥字，输出为 3 个字。

① 设输入字为 in，设 *R* 为输入字循环左移 13 比特后的值，将第 3 个密钥字 *k*′（强置成奇数）与 *R* 相乘（⊛为模 $2^{32}$ 乘），再循环左移 5 位。

② 设输入字 in 与第 1 个密钥字 *k* 的和（模 $2^{32}$ 加）为 *M*，取 *M* 的低 9 比特作为 *S* 盒的指针，取得的字为 *L*。

③ 把 *R* 的低 5 比特看成循环左移比特数（0～31 比特），将 *M* 按此数循环左移，作为第 2 个输出字 out2。

④ 将 *R* 再循环左移 5 比特作为第 3 个输出字 out3。

⑤ 将 *R* 逐比特模 2 加（即逐比特异或）到 *L* 中，再将 *L* 循环左移，以 *R* 的低 5 比特作为左移的位数(0～31 位)，变化后的结果作为第 1 个输出字 out1。

表示成伪代码形式，以上步骤可写成：

```
E 函数(输入：in,key1,key2)
设 L,M,R 为三个临时变量
M=in⊞key1
R=(in<<13)⊠key2（key2 必须置成奇数）
m=M 的低 9 比特
L=S[m]（在 S 盒查表）
R=R<<5
r=R 的低 5 比特（作为循环左移比特数）
M=M<<r
L=L⊕R
R=R<<5
L=L⊕R
r=R 的低 5 比特（作为循环左移比特数）
L=L<<r
输出(L,M,R)
```

（3）后向混乱部分。后向混乱部分相当于前向混乱部分的脱密，只是数据处理的次序不同，如图 1.40.5 所示。

与前向混乱一样，后向混乱中的每圈也使用 1 个字（$D[0]$）去修改另外 3 个字（$D[1]$、$D[2]$和 $D[3]$），设进入本部分的字的 4 个字节为 $D[0]=(b_3,b_2,b_1,b_0)$，用 $b_0$、$b_2$ 作为地址取 $S_1$，用 $b_1$、$b_3$ 作为地址取 $S_0$，将取得字来改变 $D[1]$、$D[2]$、$D[3]$，具体如下：

对于 $i$=0～7，进行如下变换：

如果 $i$=2 或 6，则

$$D[0]=D[0]-D[3]$$

如果 $i$=3 或 7，则

$$D[0]=D[0]-D[1]$$
$$D[1]=D[1]\oplus S_1[b_0]$$
$$D[2]=D[2]-S_0[b_3]$$
$$D[3]=D[3]-S_1[b_2]$$
$$D[3]=D[3]\oplus S_0[b_1]$$

再将 $D[0]$循环左移 24 比特，即

$$D[0]=D[0]<<24$$

将 $D[\ ]$循环右移 1 个字，即

$$(D[3],D[2],D[1],D[0])\leftarrow(D[0],D[3],D[2],D[1])$$

以上变换进行 8 圈后，再减去密钥字，即

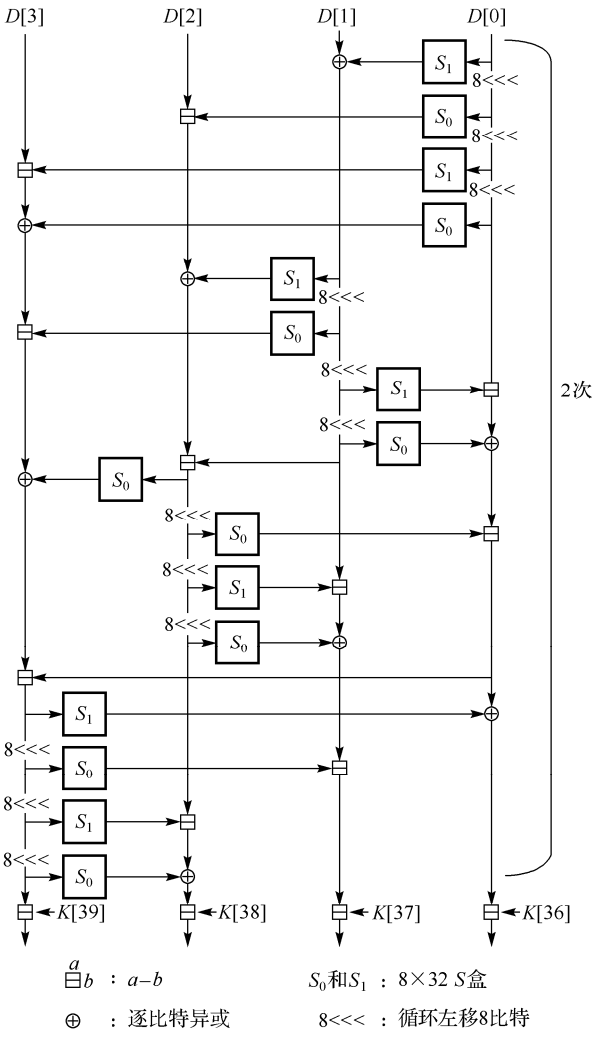



图 1.40.5　后向混乱部分的结构

$$D[i]=D[i]-K[36+i],\quad i=0,1,2,3$$

以上是加密过程，脱密是加密的逆过程，这里不再赘述，有兴趣的读者可参阅附录 B 中的伪代码。

### 3．密钥扩展

密钥扩展的框架结构如图 1.40.6 所示。

密钥扩展的目的是把 $n$（$4\leqslant n\leqslant 39$）个密钥字扩展成 40 个密钥字，并保证在加密过程用于乘法（模 $2^{32}$ 乘）的密钥字具有以下性质：

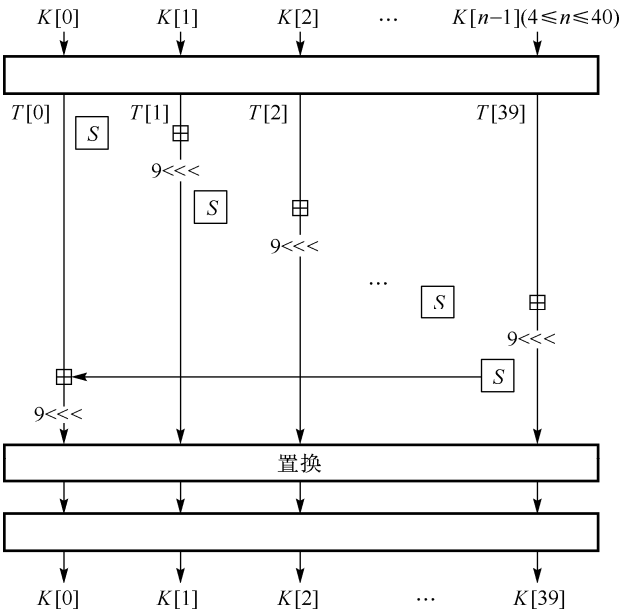



图 1.40.6　密钥扩展的框架结构

- 用于乘法的密钥字的最低 2 比特为 1；
- 这些密钥字中不出现连续的 10 个 0 或 10 个 1。

扩展过程包括 4 个步骤，具体如下。

（1）用线性变换将原始含 $n$ 个字的密钥字 $K[\ ]$扩展成 47 个临时字并存储到 $T[i]$，$i$=–7,–6,⋯,1,2,⋯,39，使最前面的 7 个字 $T[-7],\cdots,T[-1]$等于 $S$ 盒的前 7 个字，即：

$$T[i]=S[i+7],\quad i=-7,-6,\cdots,-1$$

$T[0]$到 $T[38]$用下面式子赋值：

$$T[i]=(T[i-7]\oplus T[i-2])<<<3\oplus(K[i \bmod n]\oplus i),\quad i=0,1,\cdots,38$$

$$T[39]=n$$

将最后一个临时字 $T[39]$设置成 $n$，其目的是为了避免两个不同长度的密钥扩展成相同的 40 个密钥字。如果 $T[39]$不同，就能扩展成不同的 40 个密钥字。例如，原始密钥为 4 个字的全零密钥和原始密钥为 6 个字的全零密钥就能扩展成不同的密钥。

（2）用 7 圈 Feistel1 型混乱 $T[\ ]$，即：

$$T[i]=(T[i]\boxplus S[T[i-1]\text{的低 9 比特}])<<<9,\quad i=0,1,\cdots,39$$

$$T[0]=(T[0]\oplus S[T[39]\text{低 9 比特}])<<<9$$

（3）将字重排，以便使部分相邻的密钥字拉开距离。

$$K[7i \bmod 40]=T[i], \quad i=0,1,\cdots,39$$

（4）对其中用于乘法的 16 个字进行检测，它们是 $K[5],K[7],\cdots,K[35]$，如果它们都不“弱”，则检测完毕。一个密钥字 $K[j]$是“弱”的，是指将 $K[j]$最低 2 比特置 1 后，这个字包含连续 10 个 0 或 10 个 1。因此，要对 $K[5],K[7],\cdots,K[35]$进行以下处理：

① 记录 $K[i]$的最低 2 比特 $j=K[i]\wedge 3$，考察最低 2 比特位置 1 后的字 $W=K[i]\vee 3$。

② 将 $W$ 表示成“0”“1”游程长的形式，如 $W=0^3 1^{13} 0^{12} 1001$（这里 $0^i$、$1^i$ 表示连续 $i$ 个 0 或 1），构造 $M$，使得 $W$ 中连续多于 10 个 0 或 1 的对应比特置为 1。对于上例的 $W$，构造 $M=0^3 1^{25} 0^4$，然后在连续 0 或 1 的两端置 0，即在第 4、15、16 和 28 位为 0，得 $M=0^4 1^{11} 001^{10} 0^5$。

③ 取 $B[\ ]=\{0xa4a8d57b,0x5b5d193b,0xc8a8309b,0x73f9a978\}$，这是 $S$ 盒中第 265～268 个元素。令

$$P=B[j]<<(K[i+3]\text{低 5 比特}), \quad j=K[i]\wedge 3$$

$$K[i]=W\oplus(P\wedge M), \quad i=5,7,\cdots,35$$

## 1.41 Rijndael 加密算法

Rijndael 加密算法是 AES 的候选算法之一，它是由比利时的 Vincent Rijmen 和 Joan Daeman 于 1998 年提交的，其结构与 SQUARE 加密算法完全相同，不属于 Feistel 结构。Rijndael 加密算法的分组长度和密钥长度都可以为 128、192、256 比特，迭代层数由分组长度和密钥长度共同决定。

将一个分组记成 $N_b$ 个 4 字节的字，每个 4 字节的字按字节顺序放在一列中，这样每个分组就排成了一个 4 行 $N_b$ 列的字节表。例如，当分组长度为 192 比特时，$N_b=6$，因此可以排成一个如表 1.41.1 所示的 4 行 6 列的字节表。

表 1.41.1　4 行 6 列字节表

| | | | | | |
|---|---|---|---|---|---|
| $a_{00}$ | $a_{10}$ | $a_{20}$ | $a_{30}$ | $a_{40}$ | $a_{50}$ |
| $a_{01}$ | $a_{11}$ | $a_{21}$ | $a_{31}$ | $a_{41}$ | $a_{51}$ |
| $a_{02}$ | $a_{12}$ | $a_{22}$ | $a_{32}$ | $a_{42}$ | $a_{52}$ |
| $a_{03}$ | $a_{13}$ | $a_{23}$ | $a_{33}$ | $a_{43}$ | $a_{53}$ |

Rijndael 加密算法的层函数是指以整个字节表、单个字节、行或列为单位进行的字节代替、行移位、列混合，以及与层密钥进行模 2 加四次变换。

同样，将密钥分成 $N_k$ 个 4 字节的字，则迭代层数 $N_r$ 由 $N_b$ 和 $N_k$ 决定，如表 1.41.2 所示。

表 1.41.2　$N_r$ 与 $N_b$ 和 $N_k$ 的关系

| $N_r$ | $N_b$=4 | $N_b$=6 | $N_b$=8 |
|---|---|---|---|
| $N_k$=4 | 10 | 12 | 14 |
| $N_k$=6 | 12 | 12 | 14 |
| $N_k$=8 | 14 | 14 | 14 |

Rijndael 加密算法首先将明文分组与第一个层密钥进行模 2 加，然后对 $N_r$–1 层进行同样的变换，$N_r$–1 层的层函数由四个变换组成，即

```
Round(State,Roundkey)
{
    Bytesub(State);
    Shiftrow(Satae);
    Mixcolumn(State);
    Addroundkey(State,Roundkey);
}
```

注意：最后一层的层函数没有 Mixcolumn(State)变换。

下面依次介绍这四个变换：

（1）Bytesub 是一个线性变换，它可通过一个仿射变换对字节表中的每个字节进行一次字节代替，生成一个新的字节并放在字节表的同一个位置上。假设输入字节 $x=(x_0,x_1,\cdots,x_7)$，输出字节 $y=(y_0,y_1,\cdots,y_7)$，则 Bytesub 变换为 $y=m(x)\cdot x+c$，其中 $m(x)=x^7+x^6+x^5+x^4+1$，是 $GF(2^8)$ 的一个可逆多项式。

用矩阵表示的 Bytesub 变换为：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

在具体的实现过程中，Bytesub 变换可以用一个代替表来完成。

（2）Shiftrow 变换是一个行移位变换，它对字节表中的第 2、3、4 行独立地进行行移位变换，每行的移位字节数与分组长度有关，如表 1.41.3 所示。

（3）Mixcolumn 变换可对字节表中的每列进行列混合，变换关系为：

$$\begin{pmatrix} b_{j0} \\ b_{j1} \\ b_{j2} \\ b_{j3} \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} a_{j0} \\ a_{j1} \\ a_{j2} \\ a_{j3} \end{pmatrix}$$

（4）Addroundkey 变换可对层密钥与字节表进行模 2 加，层密钥的长度等于输入分组的长度。

表 1.41.3　每行的移位字节数与分组长度的关系

| $N_b$ | $c_2$ | $c_3$ | $c_4$ |
|---|---|---|---|
| 4 | 1 | 2 | 3 |
| 6 | 1 | 2 | 3 |
| 8 | 1 | 3 | 4 |

整个算法共用 $N_r$+1 个层密钥，层密钥的生成方法如下：

（1）将密钥扩展成 $N_b(N_r+1)$个 4 字节的字并存放在数组 $w[N_b(N_r+1)]$中，当 $N_k \leqslant 6$ 和 $N_k=8$ 时扩展方法稍有差异。

当 $N_k \leqslant 6$ 时，伪代码如下：

```
for(i=0;i<Nk;i++)  w[i]=k[i];
for(j=Nk;j<Nb*(Nr+1);j+=Nk)
{
    w[j]=w[j-Nk]⊕Bytesub(Rotl(w[j-1]))⊕Rcon[j/Nk];
    for(i=1;i<Nk  &&  i+j<Nb(Nr+1);i++)
    w[i+j]=w[i+j-Nk]⊕w[i+j-1];
}
```

当 $N_k=8$ 时，伪代码如下：

```
for(i=0;i<Nk;i++)  w[i]=k[i];
for(j=Nk;j<Nb*(Nr+1);j+=Nk)
{
    w[j]=w[j-Nk]⊕Bytesub(Rotl(w[j-1]))⊕Rcon[j/Nk];
    for(i=1;i<4;i++)
    w[i+j]=w[i+j-Nk]⊕w[i+j-1];
    w[j+4]=w[j+4-Nk]⊕Bytesub(w[j+3]);
    for(i=5;i<Nk;i++)
    w[i+j]=w[i+j-Nk]⊕w[i+j-1];
}
```

上面伪代码中 Bytesub 就是前面提到的 Bytesub 变换，它可依次对 4 个字节进行字节代替；Rotl 可对每个字中的 4 个字节进行循环左移位，每次移位 1 个字节；Rcon(*i*)是一个独立于 $N_k$ 的层常数，它如下定义：

```
Rcon[i]=(RC[i], '00', '00', '00')
```

其中，

```
RC[0]='01'
RC[i]=xtime[Rcon[i-1]]
```

xtime(*a*)表示 $a \times x \bmod m(x)$，$m(x)=x^8+x^4+x^3+x+1$，$m(x)$的系数用十六进制表示为 0x11B，因此 xtime(*a*)表示将 *a* 左移 1 位，再与 0x1B 进行模 2 加。

## 1.42　Serpent 加密算法

Serpent 加密算法是 15 种 AES 候选加密算法之一，由英国的 R. Anderson、以色列的 E. Biham 和挪威的 L. Knudsen 联合提出。当时认为，AES 标准将用来保护今后几十年的大量金融事务文档、健康记录和政府信息，而对其评议和审查时间却较短，故不适宜采用新颖而未经充分分析检验的新思想，所以选择了较为稳妥、谨慎的设计思想。Serpent 加密算法以模 2 加层密钥、*S* 盒非线性代替和线性变换作为层变换，这与 DES 密码算法是类似的，它们都是以 SP（Substitution & Permutation）网络来构建层变换的分组密码的，但 Serpent 加密算法的每层变换可对整个 128 比特的输入数据同时并行地进行计算，这与 DES 不同，所以它不是 Feistel 结构的分组密码。

Serpent 加密算法是在用比特分割法实现 DES 计算这一思路的启示下提出的，其每层采用 32 个相同的 4 比特到 4 比特映射的非线性 *S* 盒，在实际计算时，将每个 *S* 盒的 4 比特分割开，放入 4 个不同的 32 比特字，使 32 个 *S* 盒可以同时进行计算，以提高加/脱密速度。32 层共用了 8 个不同的 *S* 盒，即 $S_0 \sim S_7$。

Serpent 加密算法由 32 层变换构成，每层变换由模 2 加层密钥、*S* 盒非线性代替和线性变换组成，其中第 32 层的线性变换由模 2 加 128 比特的出口密钥所代替。

层密钥的生成很简单。将 256 比特的密钥分成 8 个 32 比特的字，记为 $K_0,K_1,\cdots,K_7$，对于 $i=8,9,\cdots,131$，计算：

$$K_i=(K_{i-8} \oplus K_{i-5} \oplus K_{i-3} \oplus K_{i-1} \oplus \varphi \oplus i)<<11 \qquad (1\text{-}42\text{-}1)$$

式中，$\phi$=0x9e3779b9，$S_3(K_0 \sim K_3)$、$S_2(K_4 \sim K_7)$、$S_1(K_8 \sim K_{11})$、$S_0(K_{12} \sim K_{15})$、$S_7(K_{16} \sim K_{19})$、…、$S_4(K_{124} \sim K_{127})$分别作为 1～32 层的层密钥，$S_3(K_{128} \sim K_{131})$作为出口密钥。

Serpent 加密算法可用下列步骤来描述。

（1）初始置换 IP。

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 32 | 64 | 96 | 1 | 33 | 65 | 97 | 2 | 34 | 66 | 98 | 3 | 35 | 67 | 99 |
| 4 | 36 | 68 | 100 | 5 | 37 | 69 | 101 | 6 | 38 | 70 | 102 | 7 | 39 | 71 | 103 |
| 8 | 40 | 72 | 104 | 9 | 41 | 73 | 105 | 10 | 42 | 74 | 106 | 11 | 43 | 75 | 107 |
| 12 | 44 | 76 | 108 | 13 | 45 | 77 | 109 | 14 | 46 | 78 | 110 | 15 | 47 | 79 | 111 |
| 16 | 48 | 80 | 112 | 17 | 49 | 81 | 113 | 18 | 50 | 82 | 114 | 19 | 51 | 83 | 115 |
| 20 | 52 | 84 | 116 | 21 | 53 | 85 | 117 | 22 | 54 | 86 | 118 | 23 | 55 | 87 | 119 |
| 24 | 56 | 88 | 120 | 25 | 57 | 89 | 121 | 26 | 58 | 90 | 122 | 27 | 59 | 91 | 123 |
| 28 | 60 | 92 | 124 | 29 | 61 | 93 | 125 | 30 | 62 | 94 | 126 | 31 | 63 | 95 | 127 |

将输入的 128 比特自左至右的编号 $i$ 为 0～127，其中第 $i$ 个位置的值为 $j$ 表示经初始置换 IP 后输出的第 $i$ 个位置的值为输入第 $j$ 个位置上的值。例如，第 1 个位置上的值为 32，表示输出的第 1 个位置的值来自输入的第 32 个位置的值。

（2）进行 32 层变换，每层变换由以下三步组成。

① 模 2 加层密钥。将 128 比特的输入数据分成 4 个 32 比特的字，分别与层密钥模 2 加。

② 将模 2 加层密钥后的 128 比特分成 32 个 4 比特的分组，0～3 比特，4～7 比特，…，124～127 比特各为一组，分组经过 32 个相同的 $S$ 盒。Serpent 加密算法共有 8 个不同的 $S$ 盒，即 $S_0$～$S_7$，第 1～32 层反复循环使用 $S_0$～$S_7$，即第 1、9、17、25 层使用 $S_0$，第 2、10、18、26 层用 $S_1$ 盒，…，第 8、16、24、32 层用 $S_7$。8 个 $S$ 盒为：

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_0$: | 3 | 8 | 15 | 1 | 10 | 6 | 5 | 11 | 14 | 13 | 4 | 2 | 7 | 0 | 9 | 12 |
| $S_1$: | 15 | 12 | 2 | 7 | 9 | 0 | 5 | 10 | 1 | 11 | 14 | 8 | 6 | 13 | 3 | 4 |
| $S_2$: | 8 | 6 | 7 | 9 | 3 | 12 | 10 | 15 | 13 | 1 | 14 | 4 | 0 | 11 | 5 | 2 |
| $S_3$: | 0 | 15 | 11 | 8 | 12 | 9 | 6 | 3 | 13 | 1 | 2 | 4 | 10 | 7 | 5 | 14 |
| $S_4$: | 1 | 15 | 8 | 3 | 12 | 0 | 11 | 6 | 2 | 5 | 4 | 10 | 9 | 14 | 7 | 13 |
| $S_5$: | 15 | 5 | 2 | 11 | 4 | 10 | 9 | 12 | 0 | 3 | 14 | 8 | 13 | 6 | 7 | 1 |
| $S_6$: | 7 | 2 | 12 | 5 | 8 | 4 | 6 | 11 | 14 | 9 | 1 | 15 | 13 | 3 | 10 | 0 |
| $S_7$: | 1 | 13 | 15 | 0 | 14 | 8 | 2 | 11 | 7 | 4 | 12 | 10 | 9 | 3 | 5 | 6 |

$S$ 盒的逆为：

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inv$S_0$: | 13 | 3 | 11 | 0 | 10 | 6 | 5 | 12 | 1 | 14 | 4 | 7 | 15 | 9 | 8 | 2 |
| Inv$S_1$: | 5 | 8 | 2 | 14 | 15 | 6 | 12 | 3 | 11 | 4 | 7 | 9 | 1 | 13 | 10 | 0 |
| Inv$S_2$: | 12 | 9 | 15 | 4 | 11 | 14 | 1 | 2 | 0 | 3 | 6 | 13 | 5 | 8 | 10 | 7 |
| Inv$S_3$: | 0 | 9 | 10 | 7 | 11 | 14 | 6 | 13 | 3 | 5 | 12 | 2 | 4 | 8 | 15 | 1 |
| Inv$S_4$: | 5 | 0 | 8 | 3 | 10 | 9 | 7 | 14 | 2 | 12 | 11 | 6 | 4 | 15 | 13 | 1 |
| Inv$S_5$: | 8 | 15 | 2 | 9 | 4 | 1 | 13 | 14 | 11 | 6 | 5 | 3 | 7 | 12 | 10 | 0 |
| Inv$S_6$: | 15 | 10 | 1 | 13 | 5 | 3 | 6 | 0 | 4 | 9 | 14 | 7 | 2 | 12 | 8 | 11 |
| Inv$S_7$: | 3 | 0 | 6 | 13 | 9 | 14 | 15 | 8 | 5 | 12 | 11 | 7 | 10 | 1 | 4 | 2 |

（3）进行线性变换（LT）。

{16 52 56 70 83 94 105} {72 114 125} { 2 9 15 30 76 84 126} {36 90 103}

```
{20 56 60  74  87  98 109} { 1  76 118} { 2  6  13  19  34  80  88}  {40  94 107}
{24 60 64  78  91 102 113} { 5  80 122} { 6 10  17  23  38  84  92}  {44  98 111}
{28 64 68  82  95 106 117} { 9  84 126} {10 14  21  27  42  88  96}  {48 102 115}
{32 68 72  86  99 110 121} { 2  13  88}  {14 18  25  31  46  92 100} {52 106 119}
{36 72 76  90 103 114 125} { 6  17  92}  {18 22  29  35  50  96 104} {56 110 123}
{ 1 40 76  80  94 107 118} {10  21  96}  {22 26  33  39  54 100 108} {60 114 127}
{ 5 44 80  84  98 111 122} {14  25 100}  {26 30  37  43  58 104 112} { 3 188   }
{ 9 48 84  88 102 115 126} {18  29 104}  {30 34  41  47  62 108 116} { 7 122   }
{ 2 13 52  88  92 106 119} {22  33 108}  {34 38  45  51  66 112 120} {11 126   }
{ 6 17 56  92  96 110 123} {26  37 112}  {38 42  49  55  70 116 124} { 2  15  76}
{10 21 60  96 100 114 127} {30  41 116}  { 0 42  46  53  59  74 120} {10  23  84}
{ 3 14 25 100 104 118   }  {34  45 120}  { 4 46  50  57  63  78  12}  {10  23  84}
{ 7 18 29 104 108 122   }  {38  49 124}  { 0  8  50  54  61  67  82}  {14  27  88}
{11 22 33 108 112 126   }  { 0  42  53}  { 4 12  54  58  65  71  86}  {18  31  92}
{ 2 15 26  37  76 112 116} { 4  46  57}  { 8 16  58  62  69  75  90}  {22  35  96}
{ 6 19 30  41  80 116 120} { 8  50  61}  {12 20  62  66  73  79  94}  {26  39 100}
{10 23 34  45  84 120 124} {12  54  65}  {16 24  66  70  77  83  98}  {30  43 104}
{ 0 14 27  38  49  88 124} {16  58  69}  {20 28  70  74  81  87 102}  {34  47 108}
{ 0  4 18  31  42  53  92}  {20  62  73}  {24 32  74  78  85  91 106}  {38  51 112}
{ 4  8 22  35  46  57  96}  {24  66  77}  {28 36  78  82  89  95 110}  {42  55 116}
{ 8 12 26  39  50  61 100} {28  70  81}  {32 40  82  86  93  99 114}  {46  59 120}
{12 16 30  43  54  65 104} {32  74  85}  {36 90 103 118          }  {50  63 124}
{16 20 34  47  58  69 108} {36  78  89}  {40 94 107 122          }  { 0  54  67}
{20 24 38  51  62  73 112} {40  82  93}  {44 98 111 126          }  { 4  58  71}
{24 28 42  55  66  77 116} {44  86  97}  { 2 48 102 115          }  { 8  62  75}
{28 32 46  59  70  81 120} {48  90 101}  { 6 52 106 119          }  {12  66  79}
{32 36 50  63  74  85 124} {52  94 105}  {10 56 110 123          }  {16  70  83}
{ 0 36 40  54  67  78  89}  {56  98 109}  {14 60 114 127          }  {20  74  87}
{ 4 40 44  58  71  82  93}  {60 102 113}  { 3 18  72 114 118 125   }{24  78  91}
{ 8 44 48  62  75  86  97}  {64 106 117}  { 1  7  22  76 118 122   }  {28  82  95}
{12 48 52  66  79  90 101} {68 110 121}  { 5 11  26  80 122 126   }  {32  86  99}
```

LT 中第 $i$ 个位置括号中的数值表示输出的第 $i$ 比特是输入的哪些比特的模 2 加。例如，第 1 个位置括号中的数值是{72 114 125}，则表示经过 LT 后输出的第 1 比特是输入的第 72、114、125 比特的模 2 加。

LT 的逆变换为：

```
{ 53  55  72}   { 1  5  20  90    }  {   15 102} { 3 31 90              }
{ 57  69  76}   { 5  9  24  94    }  {   19 106} { 7 35 94              }
{ 61  63  80}   { 9 13  28  98    }  {   23 110} {11 39 98              }
{ 65  67  84}   {13 17  32 102    }  {   27 114} { 1  3 15  20  43 102   }
{ 69  71  88}   {17 21  36 106    }  { 1 31 118} { 5  7 19  24  47 106   }
{ 73  75  92}   {21 25  40 110    }  { 5 35 122} { 9 11 23  28  51 110   }
{ 77  79  96}   {25 29  44 114    }  { 9 39 126} {13 15 27  32  55 114   }
```

```
{ 81  83 100}   { 1 29  33  48 118} { 2 13  43} { 1 17 19  31  36  59 118}
{ 85  87 104}   { 5 33  37  52 122} { 6 17  47} { 5 21 23  35  40  63 122}
{ 89  91 108}   { 9 37  41  56 126} {10 21  51} { 9 25 27  39  44  67 126}
{ 93  95 112}   { 2 13  41  45  60} {14 25  55} { 2 13 29  31  43  48  71}
{ 97  99 116}   { 6 17  45  49  64} {18 29  59} { 6 17 33  35  47  52  75}
{101 103 120}   {10 21  49  53  68} {22 33  63} {10 21 37  39  51  56  79}
{105 107 124}   {14 25  53  57  72} {26 37  67} {14 25 41  43  55  60  83}
{  0 109 111}   {18 29  57  61  76} {30 41  71} {18 29 45  47  59  64  87}
{  4 113 115}   {22 33  61  65  80} {34 45  75} {22 33 49  51  63  68  91}
{  8 117 119}   {26 37  65  69  84} {38 49  79} {26 37 53  55  67  72  95}
{ 12 121 123}   {30 41  69  73  88} {42 53  83} {30 41 57  59  71  76  99}
{ 16 125 127}   {34 45  73  77  92} {46 57  87} {34 45 61  63  75  80 103}
{  1   3  20}   {38 49  77  81  96} {50 61  91} {38 49 65  67  79  84 107}
{  5   7  24}   {42 53  81  85 100} {54 65  95} {42 53 69  71  83  88 111}
{  9  11  28}   {46 57  85  89 104} {58 69  99} {46 57 73  75  87  92 115}
{ 13  15  32}   {50 61  89  93 108} {62 73 103} {50 61 77  79  91  96 119}
{ 17  19  36}   {54 65  93  97 112} {66 77 107} {54 65 81  83  95 100 123}
{ 21  23  40}   {58 69  97 101 116} {70 81 111} {58 69 85  87  99 104 127}
{ 25  27  44}   {62 73 101 105 120} {74 85 115} { 3 62 73  89  91 103 108}
{ 29  31  48}   {66 77 105 109 124} {78 89 119} { 7 66 77  93  95 107 112}
{ 33  35  52}   { 0 70  81 109 113} {82 93 123} {11 70 81  97  99 111 116}
{ 37  39  56}   { 4 74  85 113 117} {86 97 127} {15 74 85 101 103 115 120}
{ 41  43  60}   { 8 78  89 117 121} {   3  90}  {19 78 89 105 107 119 124}
{ 45  47  64}   {12 82  93 121 125} {   7  94}  { 0 23 82  93 109 111 123}
{ 49  51  68}   { 1 16  86  97 125} {  11  98}  { 4 27 86  97 113 115 127}
```

注意：第 32 层用模 2 加密钥 $K_{128}$～$K_{131}$ 代替 LT。

（3）输出置换 FP=$(IP)^{-1}$。

```
 0   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
64  68  72  76  80  84  88  92  96 100 104 108 112 116 120 124
 1   5   9  13  17  21  25  29  33  37  41  45  49  53  57  61
65  69  73  77  81  85  89  93  97 101 105 109 113 117 121 125
 2   6  10  14  18  22  26  30  34  38  42  46  50  54  58  62
66  70  74  78  82  86  90  94  98 102 106 110 114 118 122 126
 3   7  11  15  19  23  27  31  35  39  43  47  51  55  59  63
67  71  75  79  83  87  91  95  99 103 107 111 115 119 123 127
```

Serpent 加密算法的加/脱密过程是不同的，除了层密钥需要逆序使用，*S* 盒和线性变换（LT）都需要用它们的逆变换，所以 Serpent 加密算法不具有大多数分组密码都有的、被称为自逆（Involution）或自反转（Self-Reciprocal）的特点。

Serpent 加密算法在实现时并不进行初始置换 IP 和输出置换 FP，只需将输入的 128 比特

明文分成 4 个字，即 $X_0,X_1,X_2,X_3$，直接进行计算即可。初始置换 IP 的作用是使输入的 $X_0$～$X_3$ 不做任何计算，就可以将输入同一个 $S$ 盒的 4 比特分别置于 $X_0$～$X_3$ 的对应比特，以便按比特分割（Bit Slice）的思想同时计算 32 个相同的 $S$ 盒，所以 Serpent 加密算法实际上只需将下述层变换重复 32 次。

将第 $t$ 层的输入记为 $X_0,X_1,X_2,X_3$，将 $\mathrm{IP}(K_{4(t-1)},K_{4(t-1)+1},K_{4(t-1)+2},K_{4(t-1)+3})$记为 $\overline{K}_1,\overline{K}_2,\overline{K}_3,\overline{K}_4$，则 $t$ 层变换如下：

① $X_i=X_i\oplus\overline{K}_i$，$i$=0,1,2,3。

② 将 $X_i$ 的 32 比特分别记为 $x_{i,0},x_{i,1},\cdots,x_{i,31}$（$i$=0,1,2,3），则 $x_{0j},x_{1j},x_{2j},x_{3j}$ 就是一个 $S$ 盒的 4 个输入比特（$j$=0,1,…,31）。根据 $S$ 盒的代替关系，输出 $y_{0j},y_{1j},y_{2j},y_{3j}$，（$Y_0,Y_1,Y_2,Y_3$）就是经 $S$ 盒代替后的输出，其中 $Y_i=(y_{i,0},y_{i,1},\cdots,y_{i,31})$。由于 $S$ 盒的 4 个输入比特分别放在 4 个不同的 32 比特的字的相应位置，又因为每层所用的 32 个 $S$ 盒相同，所以 32 个 $S$ 盒的计算可以通过 4 个字之间的逻辑运算同时并行地进行。

③ 线性变换（LT）只需对 $S$ 盒的输出 $Y_0$～$Y_3$ 进行下列逻辑运算，伪代码如下：

```
Y0=Y0>>>13;
Y2=Y2>>>3;
Y1=Y1⊕Y0⊕Y2;
Y3=Y3⊕Y2⊕(Y0>>3);
Y1=Y1>>>1;
Y3=Y3>>>7;
Y0=Y0⊕Y1⊕Y3;
Y2=Y2⊕Y3⊕(Y1>>7);
Y0=Y0>>>5;
Y2=Y2>>>22;
```

其中，“>>”表示右移位，“>>>”表示循环右移位。经上述变化后的$(Y_0,Y_1,Y_2,Y_3)$为第 $t$ 层的输出，即第 $t$+1 层的输入（$t$=1,2,…,31）。上述变换的实质为：$\mathrm{IP}\{\mathrm{LT}^{-1}[\mathrm{FP}(Y)]\}$。对第 32 层不进行线性变换（LT），而是对出口密钥 $K_{128},K_{129},K_{130},K_{131}$ 进行模 2 加，模 2 加密钥后即可得到密文。

## 1.43　Twofish 加密算法

Twofish 加密算法是由美国 Coumterpane 公司的 Bruce Schneier 等提出的，该算法采用一个 16 圈的具有双映射 $F$ 函数的 Feistel 结构，使用了依赖于密钥的 8 比特到 8 比特的双映射 $S$ 盒。Twofish 加密算法的框架如图 1.43.1 所示。

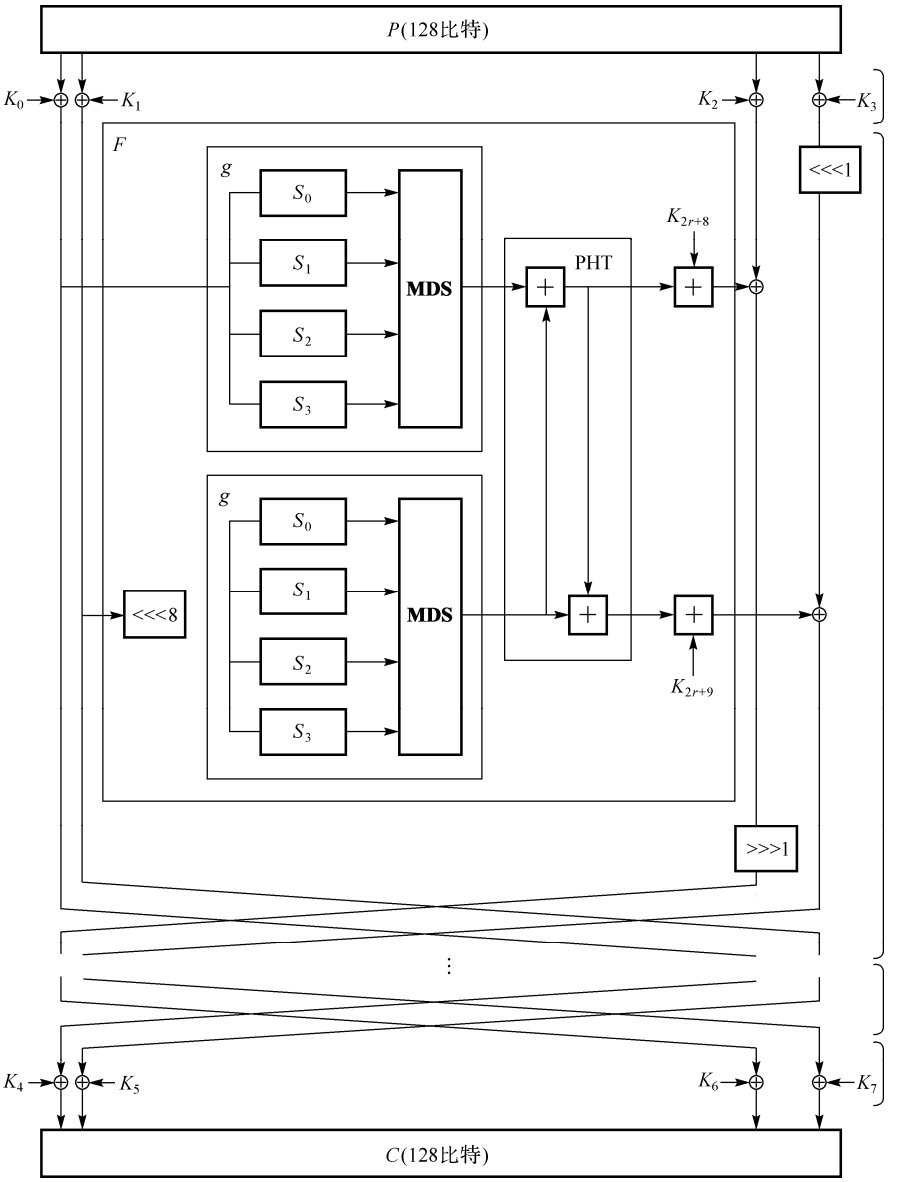



图 1.43.1　Twofish 加密算法的框架

明文 $P$（128 比特）被分为 4 个 32 比特字 $P_0$、$P_1$、$P_2$、$P_3$。

① $P_0$、$P_1$、$P_2$、$P_3$ 分别与扩展密钥字 $K_0$、$K_1$、$K_2$、$K_3$ 进行逐比特模 2 加，记为 $R_{0,i}=P_i \oplus K_i$（$i$=0,…,3）。

② $R_{0,1}$、$R_{0,2}$ 进入圈函数，前两个字 $R_{0,0}$、$R_{0,1}$ 和 $r$ 作为 $F$ 函数的输入，第三个字 $R_{0,2}$ 与 $F$ 函数的第一个输入字进行模 2 加后再循环右移 1 位，第四个字 $R_{0,4}$ 先循环左移 1 位后再与 $F$

函数输出的第 2 个字进行模 2 加，最后交换分组的左右两部分，即：

$$(F_{r,0}F_{r,1})=F(R_{r,0}R_{r,1},r)$$

$R_{r+1,0}=\mathrm{ROR}(R_{r,2}\oplus F_{r,0,1})$，$\mathrm{ROR}(x,1)$表示将字 $x$ 循环右移 1 位

$R_{r+1,1}=\mathrm{ROL}(R_{r,3},1)\oplus F_{r,1}$，$\mathrm{ROL}(x,1)$表示将字 $x$ 循环左移 1 位

$$R_{r+1,2}=R_{r,0},R_{r+1,3}=R_{r,1}$$

式中，$r$=0,1,…,15。

③ 最后一圈输出不再交换，与 4 个扩展密钥字 $K_4$、$K_5$、$K_6$、$K_7$ 进行模 2 加后可得 $C_i=R_{16,i+2}\bmod 4\oplus K_{i+4}$（$i$=0,1,2,3）。

（1）$F$ 函数。$F$ 函数的实质是一个与密钥相关的 64 比特的置换，其输入变量为 $R_0$、$R_1$ 和圈号 $r$，$r$ 用于选取适当的子密钥，$R_0$ 输入函数 $g$ 得到 $T_0$，$R_1$ 先循环左移 8 位后再输入 $g$ 得到 $T_1$。$T_0$ 和 $T_1$ 在 PHT 中组合后再加上 2 个扩展密钥，即

$$T_0=g(R_0)$$

$$T_1=g[\mathrm{ROL}(R_1,8)]$$

$$F_0=(T_0+T_1+K_{2r+8})\bmod 2^{32}$$

$$F_1=(T_0+2T_1+K_{2r+9})\bmod 2^{32}$$

式中，$F_0$ 和 $F_1$ 是 $F$ 函数输出的 2 个 32 比特的字。

（2）$g$ 函数。$g$ 函数是 $F$ 函数的核心部分，其输入是 32 比特的字 $X$，$X$ 被分成 4 个 8 比特的字节后分别进入与密钥相关的 $S$ 盒，产生 8 比特的输出，将 4 个字节看成 GF($2^8$)上 4 个元素组成的向量，与 4×4 的矩阵 **MDS** 相乘得到的 32 比特字，即 4 个 GF($2^8$)上的元素，这也是 $g$ 函数的结果。

设 $X_i$（$i$=0,…,3）为 $X$ 分成的 4 个字节，即 $X_i=\left\lfloor X/2^{8i}\right\rfloor \bmod 2^8$ $Y_i=S_i(X_i)$，$i$=0,…,3，$S_i$ 为 $S$ 盒。

$$\begin{pmatrix}Z_0\\Z_1\\Z_2\\Z_3\end{pmatrix}=\mathbf{MDS}\cdot\begin{pmatrix}y_0\\y_1\\y_2\\y_3\end{pmatrix} \tag{1-43-1}$$

$$Z=\sum_{i=0}^{3}Z_i2^{8i} \tag{1-43-2}$$

32 比特的字 $Z$ 是 $g$ 函数的结果，GF($2^8$)表示为 GF(2)[$x$]/$V(x)$，这里 $V(x)=X^8+X^6+X^5+X^3+1$，是 GF(2)上一个 8 次本原多项式，域元素

$$\alpha=\sum_{i=0}^{7}a_ix^i$$

$a_i \in$GF(2)等同于字节值，$\sum_{i=0}^{7} a_i x^i$ GF($2^8$)中的加法对应于字节的 xor（异或，即模 2 加）。

矩阵 **MDS** 的定义为：

$$\mathbf{MDS}=\begin{bmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 5B \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{bmatrix} \tag{1-43-3}$$

读者在后文中将看到，$g$ 函数可以完全由后面给出的 $h$ 函数来定义。

（3）密钥编排。密钥编排是为了将给定的密钥扩展成 40 个字的密钥 $K_0,\cdots,K_{39}$，也为 $g$ 函数中的 $S$ 盒提供了相关的密钥参数 $S$。Twofish 加密算法定义的密钥长度 $N$ 可以是 128 比特、192 比特和 256 比特，短于 256 比特的密钥都可以通过补 0 来达到上述定义的长度。例如，小于 128 比特的密钥可以通过补 0 达到 128 比特，大于 128 比特而小于 192 比特的密钥可通过补 0 成为 192 比特，而大于 192 比特小于 256 比特的密钥可以通过补 0 达到 256 比特。

定义 $k=N/64$，根据 $N$ 的三种情形（$k$=2、3 或 4），密钥 $M$ 由 $8k$ 个字节组成，即 $m_0,\cdots,m_{8k-1}$，每 4 个字节为 1 个 32 比特的字，可记为 $2k$ 个 32 比特的字。

$$M_i=\sum_{j=0}^{3} m_{4i+j}\cdot 2^{8j}, \qquad i=0,1,\cdots,2k-1 \tag{1-43-4}$$

即

$$M=(m_0,\cdots,m_{8k-1})=(M_0,M_1,\cdots,M_{2k-1}) \tag{1-43-5}$$

然后按下角标的奇偶性分为两个长度为 $k$ 的字，即

$$M_e=(m_0,m_2,\cdots,m_{2k-2}) \tag{1-43-6}$$

$$M_o=(m_1,m_3,\cdots,m_{2k-1}) \tag{1-43-7}$$

$M_e$ 和 $M_o$（由 $h$ 函数产生）是扩展密钥 $K_i$ 的重要参数。

第三个长为 $k$ 的字为向量 $\boldsymbol{S}$，由 $m_0,\cdots,m_{8k-1}$（每 8 个字节为一组）乘以 4×8 矩阵 **RS** 得出的 4 个字节组成的 32 比特的字 $S_i$，即：

$$\begin{pmatrix} S_{i,0} \\ S_{i,1} \\ S_{i,2} \\ S_{i,3} \end{pmatrix}=\mathbf{RS}\cdot\begin{pmatrix} M_{8i} \\ M_{8i+1} \\ M_{8i+2} \\ M_{8i+3} \\ M_{8i+4} \\ M_{8i+5} \\ M_{8i+6} \\ M_{8i+7} \end{pmatrix}, \qquad i=0,\cdots,k-1 \tag{1-43-8}$$

$$S_i=\sum_{j=0}^{3}S_{i,j}\cdot 2^{8j},\qquad i=0,\cdots,k-1 \tag{1-43-9}$$

式中，$\boldsymbol{S}=(S_{k-1},S_{k-2},\cdots,S_0)$，是 $g$ 函数的重要参数。

此处，对于 **RS** 乘法，GF($2^8$)用 GF(2)/$w(x)$表示 $w(x)=X^8+X^6+X^3+X^2+X$ 是 GF(2)上另一个 8 次本原多项式。**RS** 矩阵为：

$$\mathbf{RS}=\begin{bmatrix}01 & A4 & 55 & 87 & 5A & 58 & DB & 9E\\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5\\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19\\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03\end{bmatrix} \tag{1-43-10}$$

（4）扩展密钥字 $K_j$。用 $h$ 函数来定义扩展密钥字，令 $\rho=2^{24}+2^{16}+2^8+2^0$，有：

$$A_i=h(2_i\rho,M_e)$$
$$B_i=\mathrm{ROL}\{h[(2_i+1)\rho,M_0],8\}$$
$$K_{2i}=(A_i+B_i)\bmod 2^{32}$$
$$K_{2i+1}=\mathrm{ROL}[(A_i+2B_i)\bmod 2^{32},9]$$

式中，$i=0,1,\cdots,39$。

（5）$h$ 函数。$h$ 函数的逻辑框图如图 1.43.2 所示。

$h$ 函数的输入变量是 1 个 32 比特的字 $x$ 和一个有 $k$ 个字的表 $L=(L_0,\cdots,L_{k-1})$，$h$ 函数的输出为 1 个 32 比特的字 $Z$。

将 $X$ 分成 4 个字节 $X_j=\left\lfloor X/2^{8j}\right\rfloor \bmod 2^8$（$j=0,1,2,3$），每个 $L_i$ 也分成 4 个字节 $l_{i,j}=\left\lfloor L_j/2^{8j}\right\rfloor \bmod 2^8$（$j=0,1,2,3$，$i=0,\cdots,k-1$）。

令 $y_{kj}=X_j$（$j=0,\cdots,3$）,如果 $k=4$，则

$$y_{3,0}=q_1[y_{4,0}]\oplus L_{3,0}$$
$$y_{3,1}=q_0[y_{4,1}]\oplus L_{3,1}$$
$$y_{3,2}=q_0[y_{4,2}]\oplus L_{3,2}$$
$$y_{3,3}=q_1[y_{4,3}]\oplus L_{3,3}$$

如果 $k\geqslant 3$，则

$$y_{2,0}=q_1[y_{3,0}]\oplus L_{2,0}$$
$$y_{2,1}=q_1[y_{3,1}]\oplus L_{2,1}$$
$$y_{2,2}=q_0[y_{3,2}]\oplus L_{2,2}$$
$$y_{2,3}=q_0[y_{3,3}]\oplus L_{2,3}$$

再进行以下变换：

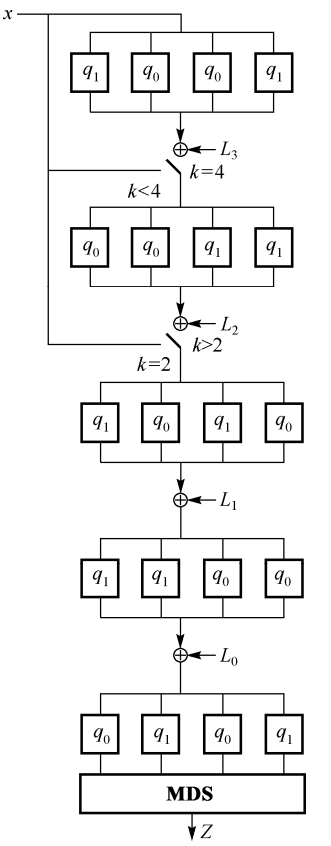



图 1.43.2　$h$ 函数的逻辑框图

$$y_0=q_1[q_0[q_0[y_{2,0}]\oplus L_{1,0}]\oplus L_{0,0}]$$
$$y_1=q_0[q_0[q_1[y_{2,1}]\oplus L_{1,1}]\oplus L_{0,1}]$$
$$y_2=q_1[q_1[q_0[y_{2,2}]\oplus L_{1,2}]\oplus L_{0,2}]$$
$$y_3=q_0[q_1[q_1[y_{2,3}]\oplus L_{1,3}]\oplus L_{0,3}]$$

式中，$q_0$和$q_1$是 8 比特置换。最后将$y_0$、$y_1$、$y_2$、$y_3$乘以 **MDS** 矩阵（已定义）即可得到函数 $h$ 的结果$z_0$、$z_1$、$z_2$、$z_3$，即

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \mathbf{MDS}\cdot\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{1-43-11}$$

$z_0$、$z_1$、$z_2$、$z_3$可组成 32 比特的字 $Z$，即

$$Z=\sum_{i=0}^{3} Z_i\cdot 2^{8i} \tag{1-43-12}$$

上述 $g(x)$实际是利用 $h(x,L)$来实现的，只是这里的 $L$ 取上面密钥编排所得的$S=(S_{k-1},S_{k-2},\cdots,S_0)$。

（6）置换 $q_0$和 $q_1$。$h$ 函数中的 $q_0$和 $q_1$是 8 比特置换，它们都由 2 个互异的 4 比特置换组成，其逻辑框图如图 1.43.3 所示。设 $x$ 为输入的 8 比特，分成 2 个 4 比特，即 $x=(a_0,b_0)$，输出为 $y=(a_4,b_4)$，$y=q_i(x)$，$i$=0 和 1，可由下面的计算而得。

$$a_1=a_0\oplus b_0$$
$$b_1=a_0\oplus \mathrm{ROR}_4(b_0,1)\oplus 8a_0 \bmod 16$$
$$(a_2,b_2)=(t_0[a_1],t_1[b_1])$$
$$a_3=a_2\oplus b_2$$
$$b_3=a_2\oplus \mathrm{ROR}_4(b_2,1)\oplus 8a_2 \bmod 16$$
$$(a_4,b_4)=(t_2[a_3],t_3[b_3])$$
$$y=16b_4+a_4$$
$$y=(a_4,b_4)$$

式中，$\mathrm{ROR}_4(b_i,1)$表示将 $b_i$的 4 比特循环右移 1 比特。当执行 $q_0$置换时，上述计算中的置换$t_1,\cdots,t_3$的取值为：

$$t_0=[817d6f320b59eca4]$$
$$t_1=[ecb81235f4a6709d]$$
$$t_2=[ba5e6d90c8f32471]$$
$$t_3=[d7f4126e9b3085ca]$$

当执行 $q_1$ 置换时，$t_0,\cdots,t_3$ 的取值为：

$t_0$=[28bdf76e31940ac5]

$t_1$=[1e2b4c376da5f908]

$t_2$=[4c75169a0ed82b3f]

$t_3$=[b951c3de647f208a]

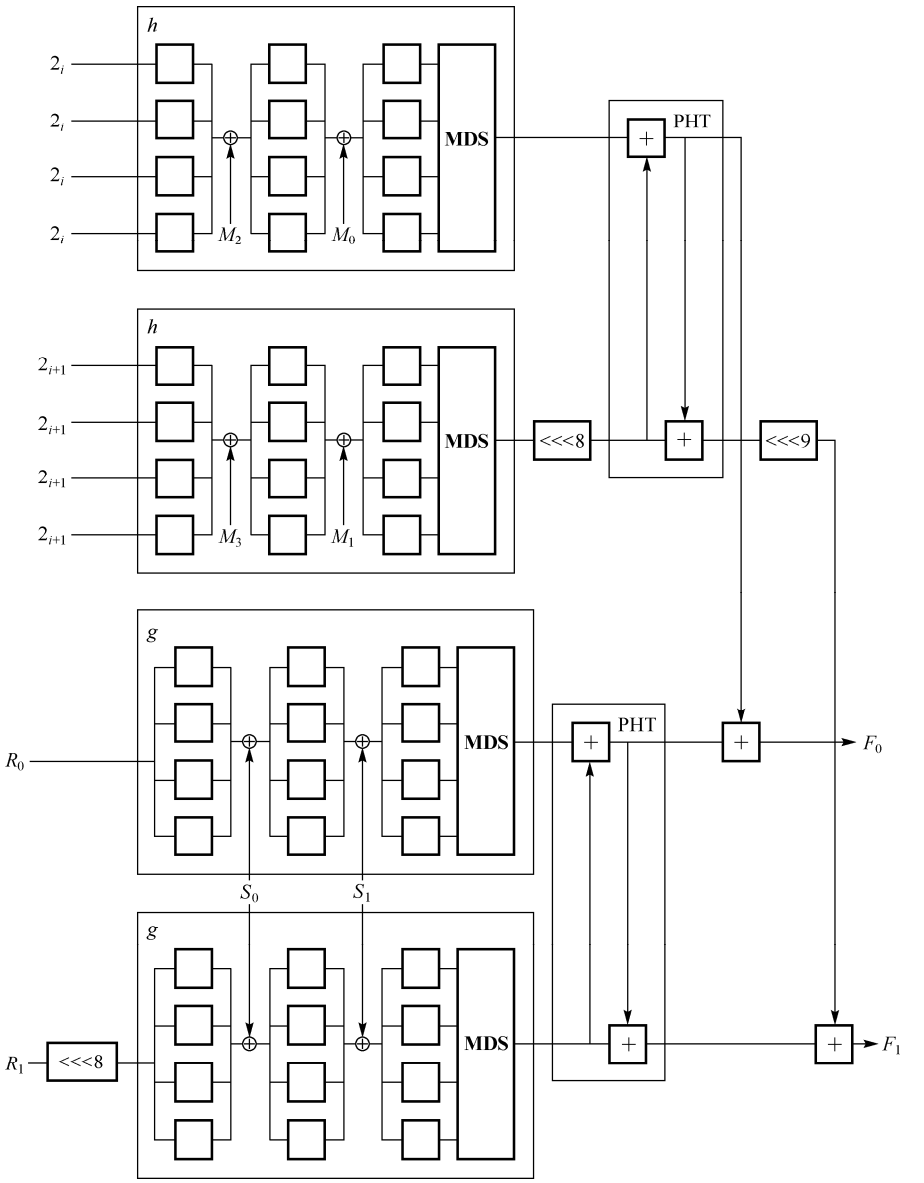



图 1.43.3　$q_0$ 和 $q_1$ 置换的逻辑框图

# 第2章

# Hash 函数

## 2.1 安全压缩算法 SHA

SHA（Secure Hash Algorithm）是美国国家标准技术局（NIST）对 MD5 算法的一种改进算法，它可将任意长度小于 264 比特的串压缩为 160 比特。SHA 是美国联邦安全压缩标准（Secure Hash Standard，SHS）规定的算法。

假设，输入消息为 $t$ 比特：

$$b_0,b_1,\cdots,b_{t-1}$$

SHA 算法由下列五步组成。

（1）填补附加比特。在消息的末尾填补 1 比特的 1，然后填补 0，直至消息的长度为模 512 余 448。

（2）附加消息长度。用 64 比特表示消息长度 $t$，放在附加比特之后。如果 $t \geq 2^{64}$，则取 $t$ 的低位 64 比特。经过该步后，消息的比特数为 512 的整数倍，其字（32 比特）数为 16 的整数倍。把这些字记为 $M[0],M[1],\cdots,M[n-1]$，$n$ 为 16 的整数倍。

前两步和 MD5 算法完全一致。

（3）初始化寄存器。SHA 算法用了 5 个 32 比特寄存器（$A$、$B$、$C$、$D$、$E$）来计算消息压缩值，5 个寄存器的初值为：

```
A: 0x67452301
B: 0xefcdab89
C: 0x98badcfe
D: 0x10325476
E: 0xc3d2e1f0
```

（4）以 16 字分组（512 比特）为单位对消息进行处理。整个算法由 4 层组成，每层包括 20 个独立的操作。每层使用了一个辅助函数和一个常数，如下所示。

$$f_0(X,Y,Z)=(X\&Y)|(\sim X\&Z)$$

$$f_1(X,Y,Z)=X\oplus Y\oplus Z$$

$$f_2(X,Y,Z)=(X\&Y)|(X\&Z)|(Y\&Z)$$

$$f_3(X,Y,Z)=X\oplus Y\oplus Z$$

$$K_0=0x5a827999$$

$$K_1=0x6ed9eba1$$

$$K_2=0x8f1bbcdc$$

$$K_3=0xca62c1d6$$

由分组内的 16 个字 $M_0,M_1,\cdots,M_{15}$ 产生 80 个 32 比特的字 $W_0$～$W_{79}$。

如果 $t$= 0～15，则令 $W_t=M_t$；如果 $t$=16～79，则令 $W_t=(W_{t-3}\oplus W_{t-8}\oplus W_{t-14}\oplus W_{t-16})<<<1$。令 AA=$A$，BB=$B$，CC=$C$，DD=$D$，EE=$E$，四层变换的伪代码如下：

```
for(i=0;i<4;i++)
for(j=0;j<20;j++)
{
    TEMP=(A<<<5)+f_i(B,C,D)+E+K_i+W_{i*20+j};
    (A,B,C,D,E)=(TEMP,A,B<<<30,C,D);
}
```

令 $A$=$A$+AA，$B$=$B$+BB，$C$=$C$+CC，$D$=$D$+DD，$E$=$E$+EE。

至此，一个 16 字分组的压缩变换就结束了，然后以现有 5 个寄存器的状态为初态对下一个 16 字分组进行变换，直至整个消息压缩完成。

（5）输出。将($A$,$B$,$C$,$D$,$E$)作为消息压缩值输出，输出顺序为从 $A$ 的最高字节到 $E$ 的最低字节。

上述五步就实现了 SHA 算法。

## 2.2 消息压缩值算法 RIPEMD-160

消息压缩值算法（也称为消息摘要算法）RIPEMD 是为欧盟 RIPE（Race Integrity Primitive Evaluation）工程设计的，它可将任意长度的消息压缩为 128 比特，后来又对 RIPEMD 算法做了修改，使其压缩长度变为 160 比特，修改后的算法称为 RIPEMD-160。

RIPEMD-160 是一个基于 MD4 的 Hash 函数，它由 10 层变换组成，每层变换包含 16 个子变换。为了保证计算速度，它采用了将 10 层变换分成左右两半，左右两半的各 5 层同时并行计算，然后将左右两半的结果进行混乱处理的办法。

设输入消息为 $t$ 个比特：

$$b_0,b_1,\cdots,b_{t-1}$$

消息压缩值算法 RIPEMD-160 由下列五步完成。

（1）填补附加比特。在消息的末尾填补 1 比特的 1，然后填补 0，直至消息的长度为模 512 余 448。

（2）附加消息长度。用 64 比特表示消息长度 $t$，放在附加比特之后。如果 $t \geqslant 2^{64}$，则取 $t$ 的低 64 比特，$t$ 的最低比特放在最左边，最高比特放在最右边。经该步后，消息的比特数为 512 的整数倍，其字（32 比特）数为 16 的整数倍。把这些字记为 $M[0],M[1],\cdots,M[n-1]$，$n$ 为 16 的整数倍。

前两步和 MD4 算法完全一样。

（3）连接值和常数定义。RIPEMD-160 算法采用 5 个 32 比特连接值 $h_1$、$h_2$、$h_3$、$h_4$、$h_5$，分别为：

$h_1$=0x67452301
$h_2$=0xefcdab89
$h_3$=0x98badcfe
$h_4$=0x10325476
$h_5$=0xc3d2e1f0

整个算法由 10 层组成，每层包括 16 个变换，每个变换使用了 3 个常数，左右两半用下角标 L 和 R 来区分，160 个变换所用的常数如下。

① 加常数。

| | | |
|---|---|---|
| $y_L[j]$=0, | $y_R[j]$=0x50a28be6, | 0⩽$j$⩽15 |
| $y_L[j]$=0x5a827999, | $y_R[j]$=0x5c4dd124, | 16⩽$j$⩽31 |
| $y_L[j]$=0x6ed9eba1, | $y_R[j]$=0x6d703ef3, | 32⩽$j$⩽47 |
| $y_L[j]$=0x8f1bbcdc, | $y_R[j]$=0x7a6d76e9, | 48⩽$j$⩽63 |
| $y_L[j]$=0xa953fd4e, | $y_R[j]$=0, | 64⩽$j$⩽79 |

② 移位常数。

$s_L$[ 0..15]={11,14,15,12,5,8,7,9,11,13,14,15,6,7,9,8}
$s_L$[16..31]={7,6,8,13,11,9,7,15,7,12,15,9,11,7,13,12}
$s_L$[32..47]={11,13,6,7,14,9,13,15,14,8,13,6,5,12,7,5}
$s_L$[48..63]={11,12,14,15,14,15,9,8,9,14,5,6,8,6,5,12}
$s_L$[64..79]={9,15,5,11,6,8,13,12,5,12,13,14,11,8,5,6}
$s_R$[ 0..15]={8,9,9,11,13,15,15,5,7,7,8,11,14,14,12,6}
$s_R$[16..31]={9,13,15,7,12,8,9,11,7,7,12,7,6,15,13,11}
$s_R$[32..47]={9,7,15,11,8,6,6,14,12,13,5,14,13,13,7,5}
$s_R$[48..63]={15,5,8,11,14,14,6,14,6,9,12,9,12,5,15,8}
$s_R$[64..79]={8,5,12,9,12,5,14,6,8,13,6,5,15,13,11,11}

③ 字位置常数。

$z_L$[ 0..15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}

$z_L$[16..31]={7,4,13,1,10,6,15,3,12,0,9,5,2,14,11,8}
$z_L$[32..47]={3,10,14,4,9,15,8,1,2,7,0,6,13,11,5,12}
$z_L$[48..63]={1,9,11,10,0,8,12,4,13,3,7,15,14,5,6,2}
$z_L$[64..79]={4,0,5,9,7,12,2,10,14,1,3,8,11,6,15,13}
$z_R$[ 0..15]={5,14,7,0,9,2,11,4,13,6,15,8,1,10,3,12}
$z_R$[16..31]={6,11,3,7,0,13,5,10,14,15,8,12,4,9,1,2}
$z_R$[32..47]={15,5,1,3,7,14,6,9,11,8,12,2,10,0,4,13}
$z_R$[48..63]={8,6,4,1,3,11,15,0,5,12,2,13,9,7,10,14}
$z_R$[64..79]={12,15,10,4,1,5,8,7,6,2,13,14,0,3,9,11}

（4）以 16 字分组为单位对消息进行处理。每层使用了一个辅助函数，每个函数都以 3 个 32 比特的字作为输入，输出为 1 个 32 比特的字，5 个辅助函数如下。

$$F(X,Y,Z)=X\oplus Y\oplus Z$$
$$G(X,Y,Z)=(X\&Y)|(\sim X\&Z)$$
$$H(X,Y,Z)=(X|\sim Y)\oplus Z$$
$$K(X,Y,Z)=(X\&Z)|(Y\&\sim Z)$$
$$I(X,Y,Z)=X\oplus (Y|\sim Z)$$

将 $M[0],M[1],\cdots,M[n-1]$按 16 字为单位进行分组。

对于第 $i=0\sim(n/16)-1$ 个分组逐一进行下列处理：

① 对于 $j=0\sim15$，令 $X[j]=M[i\times16+j]$。

② 令 $H_1=h_1$，$H_2=h_2$，$H_3=h_3$，$H_4=h_4$，$H_5=h_5$。

③ 对左右两半各 5 个寄存器赋值。

$$(A_L,B_L,C_L,D_L,E_L)=(H_1,H_2,H_3,H_4,H_5)$$
$$(A_R,B_R,C_R,D_R,E_R)=(H_1,H_2,H_3,H_4,H_5)$$

④ 左右两半同时进行五层变换。左边的变换为：

```
for(j=0;j<=15;j++)
{
    temp=A_L+((A_L+F(B_L,C_L,D_L)+X[z_L[j]]+y_L[j])<<<s_L[j])
    (A_L,B_L,C_L,D_L,E_L)=(E_L,temp,B_L,C_L<<<10,D_L)
}
for(j=16;j<=31;j++)
{
    temp=A_L+((A_L+G(B_L,C_L,D_L)+X[z_L[j]]+y_L[j])<<<s_L[j])
    (A_L,B_L,C_L,D_L,E_L)=(E_L,temp,B_L,C_L<<<10,D_L)
}
for(j=32;j<=47;j++)
```

```
{
    temp=A_L+((A_L+H(B_L,C_L,D_L)+X[z_L[j]]+y_L[j])<<<s_L[j])
    (A_L,B_L,C_L,D_L,E_L)=(E_L,temp,B_L,C_L<<<10,D_L)
}
for(j=48;j<=63;j++)
{
    temp=A_L+((A_L+K(B_L,C_L,D_L)+X[z_L[j]]+y_L[j])<<<s_L[j])
    (A_L,B_L,C_L,D_L,E_L)=(E_L,temp,B_L,C_L<<<10,D_L)
}
for(j=64;j<=79;j++)
{
    temp=A_L+((A_L+I(B_L,C_L,D_L)+X[z_L[j]]+y_L[j])<<<s_L[j])
    (A_L,B_L,C_L,D_L,E_L)=(E_L,temp,B_L,C_L<<<10,D_L)
}
```

右边的五层变换同左边的五层变换基本一致，只是用($A_R$,$B_R$,$C_R$,$D_R$,$E_R$)、$y_R[j]$、$z_R[j]$、$s_R[j]$代替($A_L$,$B_L$,$C_L$,$D_L$,$E_L$)、$y_L[j]$、$z_L[j]$、$s_L[j]$，5 个层函数按 $I$、$K$、$H$、$G$、$F$ 的顺序使用。

左右两半同时完成变换后，令 $t=H_1$，有

$$H_1=H_2+C_L+D_R$$
$$H_2=H_3+D_L+E_R$$
$$H_3=H_4+E_L+A_R$$
$$H_4=H_5+A_L+B_R$$
$$H_5=t+B_L+C_R$$

（5）输出。将($H_1$,$H_2$,$H_3$,$H_4$,$H_5$)作为消息压缩值输出，输出顺序为从 $A$ 的最低字节位到 $D$ 的最高字节位。

## 2.3　消息压缩值算法 MD2

消息压缩值算法 MD2 是由麻省理工学院（MIT）计算机科学实验室的 R. Rivest 提出的，它被互联网（Internet）电子邮件保密协议指定为消息压缩值算法（Message Digest Algorithm）之一，用于数字签名前对消息进行安全的压缩（Secure Hashing）。该算法可将任意长度的消息压缩为 128 比特，它以 16 字节为分组对消息进行压缩，而其他 Hash 函数则以 16 个 32 比特字为分组进行压缩，因此 MD2 算法比其他 Hash 函数速度要慢。

MD2 算法的安全性依赖于它使用的一个随机字节代替表 $S_0$,$S_1$,…,$S_{255}$，它是由 π=3.1415926…的数字决定的，$S$ 代替表如表 2.3.1 所示。

表 2.3.1　消息压缩值算法 MD2 的 *S* 代替表

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | d9 | 78 | f9 | c4 | 19 | dd | b5 | ed | 28 | e9 | fd | 79 | 4a | a0 | d8 | 9d |
| 1 | c6 | 7e | 37 | 83 | 2b | 76 | 53 | 8e | 62 | 4c | 64 | 88 | 44 | 8b | fb | a2 |
| 2 | 17 | 9a | 59 | f5 | 87 | b3 | 4f | 13 | 61 | 45 | 6d | 8d | 09 | 81 | 7d | 32 |
| 3 | bd | 8f | 40 | eb | 86 | b7 | 7b | 0b | f0 | 95 | 21 | 22 | 5c | 6b | 4e | 82 |
| 4 | 54 | d6 | 65 | 93 | ce | 60 | b2 | 1c | 73 | 56 | c0 | 14 | a7 | 8c | f1 | dc |
| 5 | 12 | 75 | ca | 1f | 3b | be | e4 | d1 | 42 | 3d | d4 | 30 | a3 | 3c | b6 | 26 |
| 6 | 6f | bf | 0e | da | 46 | 69 | 07 | 57 | 27 | f2 | 1d | 9b | bc | 94 | 43 | 03 |
| 7 | f8 | 11 | c7 | f6 | 90 | ef | 3e | e7 | 06 | c3 | d5 | 2f | c8 | 66 | 1e | d7 |
| 8 | 08 | e8 | ea | de | 80 | 52 | ee | f7 | 84 | aa | 72 | ac | 35 | 4d | 6a | 2a |
| 9 | 96 | 1a | d2 | 71 | 5a | 15 | 49 | 74 | 4b | 9f | d0 | 5e | 04 | 18 | a4 | ec |
| a | c2 | e0 | 41 | 6e | 0f | 51 | cb | cc | 24 | 91 | af | 50 | a1 | f4 | 70 | 39 |
| b | 99 | 7c | 3a | 85 | 23 | b8 | b4 | 7a | fc | 02 | 36 | 5b | 25 | 55 | 97 | 31 |
| c | 2d | 5d | fa | 98 | e3 | 8a | 92 | ae | 05 | df | 29 | 10 | 67 | 6c | ba | c9 |
| d | d3 | 00 | e6 | cf | e1 | 9e | a8 | 2c | 63 | 16 | 01 | 3f | 58 | e2 | 89 | a9 |
| e | 0d | 38 | 34 | 1b | ab | 33 | ff | b0 | bb | 48 | 0c | 5f | b9 | b1 | cd | 2e |
| f | c5 | f3 | db | 47 | e5 | a5 | 9c | 77 | 0a | a6 | 20 | 68 | fe | 7f | c1 | ad |

表中，$S_0$=0xd9，$S_1$=0x78，…，$S_{255}$=0xad。

消息压缩值算法 MD2 由以下几步组成：

（1）在消息后面填补一些字节，使得消息长度恰为 16 字节（128 比特）的整数倍。假设消息长度为 *t* 字节，计算 $m=t\%16$，$m_1=16-m$，在消息后面添加 $m_1$ 个 0x$m_1$（$m_1$ 的十六进制数）。例如，$t$=33，$m$=1，$m_1$=15，则填补 15 个 0x15，这样得到的消息长度为（$t+m_1$）字节，16 字节分组数 *n* 为$(t+m_1)/16$。

（2）添加校验和。将 *n* 个 16 字节分组对应比特进行模 2 加，得到一个 16 字节的校验和并填加到消息的后面，这样要压缩的消息长度为 *n*+1 个 16 字节，存储在 $M_0,M_1,\cdots,M_{(n+1)16-1}$ 中。

（3）对寄存器赋初值。MD2 使用了 48 个单字节寄存器 $X_0,X_1,\cdots,X_{47}$，对寄存器赋初值的伪代码如下：

```
for(i= 0;i<=15;i++)
X_i=0
for(i=16;i<=31;i++)
X_i=M_{i-16}
for(i=32;i<=47;i++)
X_i=X_{i-32} ⊕ X_{i-16}
```

（4）以 16 字节分组为单位对消息进行处理。对第 $i$=0～*n* 个 16 字节分组逐一进行下列处理：

```
{
    t=0
    for(j=0;j<=17;j++)
    for(k=0;k<=47;k++)
    {
        t=X_k ⊕ S_t
        X_k=t,
        t=(t+j) mod 256
    }
}
```

将第 $i$+1 个 16 字节消息赋给 $X_{16}$～$X_{31}$，即

```
for(j=16;j<=31;j++)
X_j=M_{16i+j}
for(j=32;j<=47;j++)
X_j=X_{j-32} ⊕ X_{j-16}
```

（5）前 16 个寄存器 $X_0,X_1,\cdots,X_{15}$ 内的 16 个字节的值作为结果输出。

## 2.4　消息压缩值算法 MD4

消息压缩值算法 MD4 是由麻省理工学院（MIT）计算机科学实验室的 R. Rivest 于 1990 年欧洲密码年会上提出的，用于数字签名前对消息进行安全压缩（Secure Hashing）。该算法可将任意长度的消息压缩为 128 比特，然后进行数字签名。

假设输入消息为 $t$ 个比特：

$$b_0,b_1,\cdots,b_{t-1}$$

按下列五步可实现消息压缩值算法 MD4。

（1）填补附加比特。在消息的后面填补 1 比特的 1，然后填补 0，直至消息的长度为模 512 余 448。

（2）附加消息长度。用 64 比特表示消息长度 $t$，放在附加比特之后。如果 $t\geqslant 2^{64}$，则取 $t$ 的低 64 比特，最低比特放在前面，最高比特放在后面。经过该步后，消息的比特数为 512 的整数倍，其字（32 比特）数为 16 的整数倍，把这些字记为 $M[0],M[1],\cdots,M[n-1]$，$n$ 为 16 的整数倍。

（3）初始化 MD 寄存器及常数定义。MD4 用 4 个 32 比特寄存器 $A$、$B$、$C$、$D$ 计算消息压缩值，这 4 个寄存器的初值（用十六进制表示）为：

```
A: 0x67452301
```

*B*: 0xefcdab89
*C*: 0x98badcfe
*D*: 0x10325476

整个 MD4 由 3 层组成，每层包括 16 个变换，每个变换使用了 3 个常数，48 个变换所用的常数如下：

① 加常数。

```
y[j]=0,    0≤j≤15
y[j]=0x5a827999,    16≤j≤31
y[j]=0x6ed9eba1,    32≤j≤47
```

② 移位常数。

```
s[ 0..15]={ 3, 7,11,19, 3, 7,11,19, 3, 7,11,19, 3, 7,11,19}
s[16..31]={ 3, 5, 9,13, 3, 5, 9,13, 3, 5, 9,13, 3, 5, 9,13}
s[32..47]={ 3, 9,11,15, 3, 9,11,15, 3, 9,11,15, 3, 9,11,15}
```

③ 字位置常数。

```
z[ 0..15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
z[16..31]={0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15}
z[32..47]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15}
```

（4）以 16 字分组为单位对消息进行处理。每层使用了一个辅助函数，每个函数都以 3 个 32 比特的字作为输入，输出为 1 个 32 比特的字，3 个辅助函数的定义为：

$$F(X,Y,Z)=(X\&Y)|(\sim X\&Z)$$
$$G(X,Y,Z)=(X\&Y)|(X\&Z)|(Y\&Z)$$
$$H(X,Y,Z)=X\oplus Y\oplus Z$$

将 $M[0],M[1],\cdots,M[n-1]$按 16 个字为单位进行分组，对第 $i=0\sim n/16-1$ 个分组逐一进行下列处理：

① 对于 $j=0\sim15$，令 $X[j]=M[i\times16+j]$。

② 令 AA=*A*，BB=*B*，CC=*C*，DD=*D*。

③ 进行 3 层 48 个变换，伪代码如下：

```
for(j=0;j<=15;j++)
{
    temp=(A+F(B,C,D)+X[z[j]]+y[j]) <<<s[j]
    (A,B,C,D)=(D,temp,B,C)
}
for(j=16;j<=31;j++)
```

```
{
    temp=(A+G(B,C,D)+X[z[j]]+y[j]) <<<s[j]
    (A,B,C,D)=(D,temp,B,C)
}
for(j=32;j<=47;j++)
{
    temp=(A+H(B,C,D)+X[z[j]]+y[j]) <<<s[j]
    (A,B,C,D)=(D,temp,B,C)
}
```

令 $A=A+AA$，$B=B+BB$，$C=C+CC$，$D=D+DD$，其中“+”表示模 $2^{32}$ 加。

（5）输出。将(*A*,*B*,*C*,*D*)作为消息压缩值输出，输出顺序为从 *A* 的最低字节位到 *D* 的最高字节位。

上述五步就实现了消息压缩值算法 MD4。

## 2.5　消息压缩值算法 MD5

消息压缩值算法 MD5 也是由麻省理工学院（MIT）计算机科学实验室的 R. Rivest 提出的，它被互联网（Internet）电子邮件保密协议（PEM）指定为消息压缩值算法（Message Digest Algorithm）之一，用于数字签名前对消息进行安全的压缩（Secure Hashing）。该算法可将任意长度的消息压缩为 128 比特，然后进行数字签名。R. Rivest 在 1990 年的欧洲密码年会上提出了 MD4 算法，MD5 算法是 MD4 算法的改进，比 MD4 稍慢，但其安全性提高了很多。美国国家标准和技术局（NIST）对 MD5 算法又做了修改，修改后的算法称为 SHA，作为美国联邦安全压缩标准（Secure Hash Standard，SHS）规定的算法。SHA 输入消息的长度小于或等于 264 比特，输出的 160 比特作为压缩值。

MD5 算法在 32 位计算机上的计算速度非常快，不需要任何大的代替表，实现该算法的代码是非常紧凑的。我们知道，DES 可用来计算消息压缩值，同样，有的消息压缩值算法也可用来构建分组密码。因为 MD5 算法的计算速度快，有人提议以它为基础来构建快速软件分组密码体制。

在基于开放网络互联的应用中，MD5 算法的客体识别标志为：

```
MD5 OBJECT IDENTIFIER::=
{iso(1) member-body(2) US(840) rsadsi(113549)
digestalgorithm(2) 5}
```

在 X.509 类型算法识别标志中，MD5 算法的参数应为 NULL 类型。

假设输入消息为 *t* 比特：

$$b_0,b_1,\cdots,b_{t-1}$$

按下列五步可实现消息压缩值算法 MD5。

（1）填补附加比特。在消息的后面填补 1 比特的 1，然后填补 0，直至消息的长度为模 512 余 448。

（2）附加消息长度。用 64 比特表示消息长度 $t$，放在附加比特之后。如果 $t \geqslant 2^{64}$，则取 $t$ 的低 64 比特，$t$ 的最低比特放在最左边，最高比特放在最右边。经过该步后，消息的比特数为 512 的整数倍，其字（32 比特）数为 16 的整数倍，把这些字记为 $M[0],M[1],\cdots,M[n-1]$，$n$ 为 16 的整数倍。

前两步和 MD4 算法完全一样。

（3）初始化 MD 寄存器和常数定义。MD5 算法使用 4 个 32 比特寄存器 $A$、$B$、$C$、$D$ 来计算消息压缩值，4 个寄存器的初值（用十六进制表示）为：

```
A: 0x01234567
B: 0x89abcdef
C: 0xfedcba98
D: 0x76543210
```

MD5 算法由 4 层组成，每层包括 16 个变换，每个变换使用 3 个常数（分别为加常数、移位常数和字位置常数），64 个变换使用的常数如下。

① 64 个加常数。64 个加常数由 $2^{32}|\sin(j+1)|$生成，其中 $0 \leqslant j \leqslant 63$，以弧度为单位。

| $j$ | $y[j]$ | $j$ | $y[j]$ | $j$ | $y[j]$ | $j$ | $y[j]$ |
|---|---|---|---|---|---|---|---|
| 0 | d76aa478 | 16 | f61e2562 | 32 | fffa3942 | 48 | f4292244 |
| 1 | e8c7b756 | 17 | c040b340 | 33 | 8771f681 | 49 | 432aff97 |
| 2 | 242070db | 18 | 265e5a51 | 34 | 6d9d6122 | 50 | ab9423a7 |
| 3 | c1bdceee | 19 | e9b6c7aa | 35 | fde5380c | 51 | fc93a039 |
| 4 | f57c0faf | 20 | d62f105d | 36 | a4beea44 | 52 | 655b59c3 |
| 5 | 4787c62a | 21 | 2441453 | 37 | 4bdecfa9 | 53 | 8f0ccc92 |
| 6 | a8304613 | 22 | d8a1e681 | 38 | f6bb4b60 | 54 | ffeff47d |
| 7 | fd469501 | 23 | e7d3fbc8 | 39 | bebfbc70 | 55 | 85845dd1 |
| 8 | 698098d8 | 24 | 21e1cde6 | 40 | 289b7ec6 | 56 | 6fa87e4f |
| 9 | 8b44f7af | 25 | c33707d6 | 41 | eaa127fa | 57 | fe2ce6e0 |
| 10 | ffff5bb1 | 26 | f4d50d87 | 42 | d4ef3085 | 58 | a3014314 |
| 11 | 895cd7be | 27 | 455a14ed | 43 | 4881d05 | 59 | 4e0811a1 |
| 12 | 6b901122 | 28 | a9e3e905 | 44 | d9d4d039 | 60 | f7537e82 |
| 13 | fd987193 | 29 | fcefa3f8 | 45 | e6db99e5 | 61 | bd3af235 |
| 14 | a679438e | 30 | 676f02d9 | 46 | 1fa27cf8 | 62 | 2ad7d2bb |
| 15 | 49b40821 | 31 | 8d2a4c8a | 47 | c4ac5665 | 63 | eb86d391 |

② 64 个移位常数。

```
s[0 ..15]={ 7,12,17,22, 7,12,17,22, 7,12,17,22, 7,12,17,22}
s[16..31]={ 5, 9,14,20, 5, 9,14,20, 5, 9,14,20, 5, 9,14,20}
s[32..47]={ 4,11,16,23, 4,11,16,23, 4,11,16,23, 4,11,16,23}
s[48..63]={ 6,10,15,21, 6,10,15,21, 6,10,15,21, 6,10,15,21}
```

③ 64 个字位置常数。

```
z[0 ..15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
z[16..31]={1,6,11,0,5,10,15,4,9,14,3,8,13,2,7,12}
z[32..47]={5,8,11,14,1,4,7,10,13,0,3,6,9,12,15,2}
z[48..63]={0,7,14,5,12,3,10,1,8,15,6,13,4,11,2,9}
```

（4）以 16 字分组为单位对消息进行处理。先定义 4 个辅助函数，每个函数都以 3 个 32 比特的字作为输入，输出 1 个 32 比特的字。

$$F(X,Y,Z)=(X\&Y)|(\sim X\&Z)$$
$$G(X,Y,Z)=(X\&Z)|(Y\&\sim Z)$$
$$H(X,Y,Z)=X\oplus Y\oplus Z$$
$$I(X,Y,Z)=Y\oplus (X|\sim Z)$$

将 $M[0],M[1],\cdots,M[n-1]$按 16 个字为单位进行分组，对第 $i=0\sim n/16-1$ 个分组逐一进行下列处理。

① 对于 $j=0\sim 15$，令 $X[j]=M[i\times 16+j]$。

② 令 AA=$A$，BB=$B$，CC=$C$，DD=$D$。

③ 进行下列 4 层变换，伪代码如下：

```
for(j=0;j<=15;j++)
{
    temp = B+((A+F(B,C,D)+X[z[j]]+y[j])<<<s[j])
    (A,B,C,D)=(D,temp,B,C)
}
for(j=16;j<=31;j++)
{
    temp = B+((A+G(B,C,D)+X[z[j]]+y[j])<<<s[j])
    (A,B,C,D)=(D,temp,B,C)
}
for(j=32;j<=47;j++)
{
    temp = B+((A+H(B,C,D)+X[z[j]]+y[j])<<<s[j])
    (A,B,C,D)=(D,temp,B,C)
```

```
    }
    for(j=48;j<=63;j++)
    {
        temp = B+((A+I(B,C,D)+X[z[j]]+y[j])<<<s[j])
        (A,B,C,D)=(D,temp,B,C)
    }
```

令 $A=A+AA$，$B=B+BB$，$C=C+CC$，$D=D+DD$。

（5）输出。将(*A*,*B*,*C*,*D*)作为消息压缩值输出，输出顺序为从 *A* 的最低字节到 *D* 的最高字节。

经过上述五步即可实现消息压缩值算法 MD5。

# 第3章

# 序 列 密 码

## 3.1 A5 加密算法

A5 加密算法是 GSM 标准中规定的序列密码算法，用于在数字蜂窝移动通信系统中加密从移动站到基站之间的链路。GSM 标准中的加密算法 A5 曾有三种候选算法——“法国算法”“瑞士算法”“英国算法”。经讨论，法国人设计的算法被 GSM 标准采用，称为 A5 算法。该算法原来是保密的，但由于民间的广泛使用，使 A5 算法的大部分细节被逐步公开了。

数字蜂窝移动通信系统由交换系统（SS）、基站（BS）和移动站（MS）三大部分组成，其中交换系统又由移动业务交换中心（MSC）、验证中心（AUC）、归属位置寄存器（HLR）、漫游位置寄存器（VLR）和移动设备识别寄存器（EIR）等组成。

用户在入网注册时，由注册当局发给用户一个用户识别模块（Subscriber Identification Module，SIM）卡，并在 SIM 卡中写入国际移动用户身份标志（International Mobile Subscriber Identity，IMSI）、验证算法 A3、密钥生成算法 A8、用户 $i$ 的主密钥 $K_i$ 以及每个用户的个人身份号（Personal Identification Number，PIN）。用户的 IMSI 和 $K_i$ 同时存入 AUC，在 AUC 中也有 A3 算法和 A8 算法，还有一个伪随机数发生器，用于产生 128 比特的随机数 RAND。将 RAND 作为算法的输入，以 $K_i$ 作为密钥，经 AUC 中的 A3 算法产生验证码 SRES，经 A8 算法生成会晤密钥 SK。RAND 和由它生成的 SK、SRES 组成用户的一个三参数组，存在 HLR 中。这样的三参数组可以根据需要不断生成，在每次验证、加密时使用一个。当移动用户进入一个新的小区时，该小区的 VLR 就向 HLR 请求传送用户的三参数组，并存入新小区的 VLR。

GSM 标准采用了多种安全防护措施。用户必须将 SIM 卡插入移动电话，并输入用户的 PIN 后方能启动设备。在进行通信时，移动业务交换中心（MSC）要通过基站对用户进行验证，通过验证后用户方能进行通信。具体说，当用户开机请求接入网络时，用户所在位置的 MSC 和 VLR 通过控制信道将三参数组的 RAND 发给用户，用户的 SIM 卡在收到 RAND 后，在 SIM 卡中通过 $K_i$ 和 A3 算法生成 SRES，并传送给 MSC 和 VLR。MSC 和 VLR 将收到的 SRES 与存储的三参数组中的 SRES 进行比较，结果相同才允许接入。另一方面，在验证过程中，在 SIM 卡通过 RAND 和 $K_i$ 使用 A8 算法还生成了 SK，用于本次会晤的加密通信。基站和移动站有共同的 A5 加密算法，当 MSC 和 VLR 在启动加密时，将 SK 传送给基站，使

基站与移动设备具有共同的会晤密钥 SK。加/脱密是在基站和移动站之间的无线通信链路上进行的。在加乱过程中，SK 不断地与随机数（帧号）相结合重置 A5 乱数发生器的初态。下面先介绍 A5 加密算法。

A5 乱数序列发生器由三个较短的线性移位寄存器（LFSR）组成，三个 LFSR 分别为 19 级、22 级、23 级，共 64 级。反馈多项式均为本原多项式，即

$$f_1(x)=x^{19}+x^{18}+x^{17}+x^{14}+1$$
$$f_2(x)=x^{22}+x^{21}+x^{17}+x^{13}+1$$
$$f_3(x)=x^{23}+x^{22}+x^{19}+x^{18}+1$$

A5 加密算法的逻辑框图如图 3.1.1 所示。

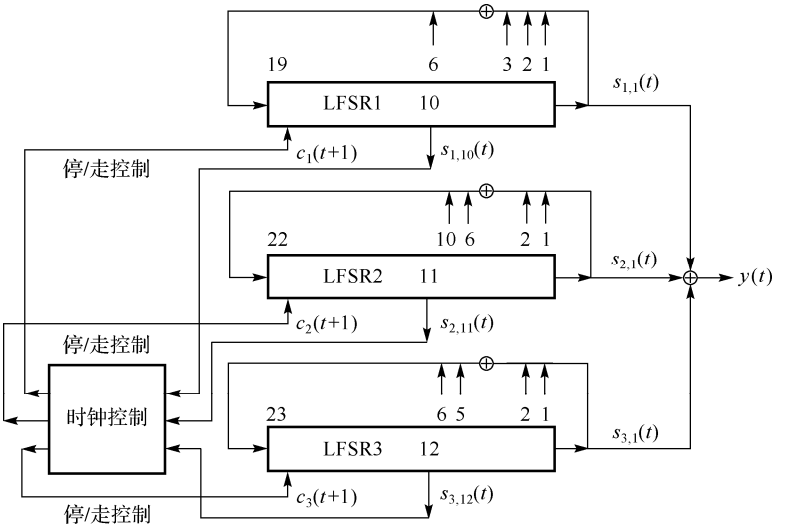



图 3.1.1　A5 加密算法的逻辑框图

将第 $i$ 个 LFSR 在 $t$ 时刻的状态记为 $S_i(t)$，即

$$S_1(t)=[s_{1,1}(t), s_{1,2}(t), \cdots, s_{1,19}(t)]$$
$$S_2(t)=[s_{2,1}(t), s_{2,2}(t), \cdots, s_{2,22}(t)]$$
$$S_3(t)=[s_{3,1}(t), s_{3,2}(t), \cdots, s_{3,23}(t)]$$

线性移位寄存器（LFSR）不规则地停/走，停/走控制信号来自三个 LFSR 的中间一级，具体为第 10、11、12 级，即 $s_{1,10}(t)$、$s_{2,11}(t)$、$s_{3,12}(t)$，将它们记为 $\tau_i(t)$（$i$=1,2,3），由它们联合控制三个 LFSR 的停/走。将三个 LFSR 在 $t$ 时刻的停/走控制信息记为 $C_i(t)$（$i$=1,2,3）。$C(t+1)=[C_1(t+1), C_2(t+1), C_3(t+1)]$是以 $\tau_1(t)$、$\tau_2(t)$、$\tau_3(t)$为输入的四值函数，当 $\tau_1(t)=\tau_2(t)=\tau_3(t)$ 时，$C_1(t+1)=C_2(t+1)=C_3(t+1)=1$，即下一拍三个 LFSR 均走；否则必有 $\tau_i(t)=\tau_j(t)\neq\tau_k(t)$，其中$\{i,j,k\}$

为{1,2,3}的一个排列，这时 $C_i(t+1)=C_j(t+1)=1$，$C_k(t+1)=0$，即下一拍第 $i$ 和 $j$ 个 LFSR 走，第 $k$ 个 LFSR 停步。由停/走规则可知，每一时刻至少有两个 LFSR 走。乱数序列 $y(t)$（$t$=0,1,…）由三个 LFSR 的输出进行模 2 加合成。

LFSR 的加乱初态由 64 比特的会晤密钥（SK）与 22 比特的随机数（帧号）相结合产生，先将 64 比特的 SK 输入三个 LFSR，为了避免 LFSR 状态为全“0”，将 LFSR 的最后一级置“1”，然后将 22 比特的随机数加入每个 LFSR 的反馈线路，LFSR 的停/走控制与加乱相同。确切地讲，每个 LFSR 先按停/走控制规则停/走，然后将随机数的相应比特分别与每个 LFSR 的最后 1 比特进行模 2 加，如此进行 22 步后，得到的三个 LFSR 状态即 LFSR 的加乱初态，亦称为消息密钥，由此开始加乱。加乱得到的前 100 比特丢弃不用（即通常所说的“空跑”），接下来的 114 比特用于加密一方的一帧消息。因为同一密钥用于全双工通信，所以接下来又丢弃 100 比特，然后用后面的 114 比特加密反向的一帧消息。下一帧再由相同的 SK 与 22 比特的随机数相结合，重置 LFSR 的加乱初态，生成下一帧的加密乱数。加乱初态的预置十分频繁，每次预置加乱初态后，只生成 428 比特的乱数，其中 228 比特用于双方的通信加密。

## 3.2　RC4 加密算法

RC4 加密算法是 R. Rivest 于 1987 年为 RSA 数据安全公司设计的专用算法之一。RC4 加密算法可看成以正整数 $n$ 为参数的一组算法，$n$ 的取值一般为 8。对于 $n$ 的不同取值，可得到不同的 RC4 加密算法。RC4 加密算法的密钥以字节为单位，共 $r$ 个字节，$r$ 可变。

RC4 加密算法的设计思想与通常的基于线性反馈移位寄存器的序列密码算法不同，它是一种基于表错乱原理的序列密码体制，与它同类的还有 ISAAC 加密算法等。这类序列密码以一个相对比较大的表为基础，在自身控制下（也可由多个表互控）慢慢变化，同时生成乱数序列。

RC4 加密算法由一个 $2^n$ 个 $n$ 比特构成的 $S$ 表和两个 $n$ 比特的指针构成，其中 $S$ 表是一个 $0\sim2^n-1$ 的排列。定义 $n$ 比特为一个字，则 RC4 加密算法所需内存量仅为 $2^n+2$ 个字，即 $n2^n+2n$ 比特。当 $n$=8 时，仅为 258 字节。将这 $2^n+2$ 个字称为内部状态，在加乱过程中，内部状态不断慢慢地变化，确切地讲是在两个指针的控制下不断地做对换，同时每一时刻输出一个字的乱数。

将 $t$ 时刻的 $S$ 表记为 $S_t=[S_t(l)]_{l=0}^{2^n-1}$，$t$ 时刻的两个指针分别记为 $i_t$ 和 $j_t$，$t$ 时刻的输出记为 $z_t$，并将 $n$ 比特的字的二进制表示和整数表示看成是同等的。

设 $i_0=j_0=0$，对于 $t\geqslant l$，RC4 加密算法的下一个状态和输出函数的定义为：

$$i_t=i_{t-1}+l$$

$$j_t=j_{t-1}+S_{t-1}(i_t)$$

$$S_t(i_t)=S_{t-1}(j_t),\qquad S_t(j_t)=S_{t-1}(i_t)$$

即交换 $S_{t-1}(j_t)$和 $S_{t-1}(i_t)$。

$$z_t=S_t[S_t(i_t)+S_t(j_t)]$$

式中，“+”为模 $2^n$ 加。输出的乱数序列以字为单位，乱数序列 $Z=(z_t)_{t=1}^{\infty}$。

$S$ 表的初态 $S_0$ 是由 0～255 顺序排列的表 $(l)_{l=0}^{2^n-1}$ 在密钥的作用下错乱生成的。错乱方法与加乱过程类似，具体如下。

令 $R_0=(l)_{l=0}^{2^n-1}$，$j_0$=0，将 $r$ 个密钥字节 $k_0,k_1,\cdots,k_{r-1}$ 多次循环反复排列，组成密钥表 $K=(k_l)_{l=0}^{2^n-1}$。对 $t$=1,2,$\cdots$,$2^n$ 计算：

$$j_t=[j_{t-1}+R_{t-1}(t-1)+k_{t-1}]\bmod 2^n$$

并交换 $R_{t-1}(t-1)$和 $R_{t-1}(j_t)$，经 $2^n$ 次对换后得到的 $R_{2^n}$，即为 $S$ 表初始值 $S_0$。

RC4 加密算法生成的乱数序列最低比特距 2 有异号优势，优势为 $15\times2^{-3n}$。识别这一统计特性所需的序列长度约为 $64^n/225$。当 $n\leqslant8$ 时，在高速信号加密中有可能用此统计特性来区分 RC4 加密算法和其他乱数发生器。

美国软件出版商协会（Software Publishers Association，SPA）和美国政府之间曾达成一项协议，给予 RC2 加密算法和 RC4 加密算法特殊的出口地位，放松了对它们的出口限制，但它们的密钥必须限于 40 比特以内。按照这一协议，若产品采用 RC2 加密算法和 RC4 加密算法，且密钥不超过 40 比特，则其出口批准手续比其他产品简单得多，只要用一组美国国家安全局提供的测试向量进行测试，得到预期结果后就可以获准出口。因此，大量的商用安全产品中使用了 RC4 加密算法，其密钥都是 40 比特。

1995 年 7 月 14 日，Hal Finney 在互联网上公布了用 Netscape 中的 RC4 加密算法（40 比特密钥）加密的一则消息（是加密的信用卡 order），提出了破译挑战。有两个研究组独立地求出了密钥，法国的一个研究组率先宣布获得成功，他们用了 120 台计算机和工作站的空闲时间，仅用 8 天就求出了 40 比特的密钥，密钥用十六进制表示就是“73 F0 96 1F 16”。1995 年 8 月 19 日又对 RC4 提出了第二次挑战，应战者通过互联网将 201 个参加者连接起来，协同进行攻击，攻击从 1995 年 8 月 24 日 18:00(GMT)开始。由于攻击软件有了很大的改进，在进行穷尽攻击的客户工作站和对搜索密钥进行分配的中心服务器之间实现了自动通信，所以只用 31.8 小时就求出了密钥“96 36 34 0D 46”。由于这次攻击只用了机器可用时间的 1/4，因此从理论上讲，这次攻击只需使用 8 小时即可求出密钥。

## 3.3 ISAAC 加密算法

ISAAC 加密算法是 R. J. Jenkins 在 1996 年召开的快速软件加密算法研讨会上提出的一种

乱数序列密码算法。R. J. Jenkins 先设计了 IA 加密算法，为解决其统计上的不平衡性，对 IA 加密算法做了修改，修改后的算法称为 IBAA 加密算法；为了进一步提高速度，避免弱加乱初态和保证周期大于 240，对算法再次做了修改，得到了 ISAAC 加密算法。

ISAAC 加密算法与 RC4 加密算法是相互独立研究设计的，但它们很相似，都以一个较大的代替表（*S* 盒）为基础，在表的逐步变化过程中生成乱数序列。ISAAC 取自 Indirection、Shift、Accumulate、Add 和 Count 五个英文单词的首字母，表示 ISAAC 加密算法是在秘密代替表的基础上，通过间接取址、移位、累加、加法和加计数器值等基本运算生成乱数序列的加密算法。

ISAAC 加密算法所用的秘密代替表是由 256 个 32 比特的字构成的，将其存放在数组 *m*[0]～*m*[255]中，每次调用 ISAAC 加密算法生成的 256 个 32 比特乱数，存放在数组 *r*[0]～*r*[255]中。调用 ISAAC 加密算法时需要 *a*、*b*、*c* 三个参数，*a*、*b* 为初始值，*c* 是调用该算法次数的计数值，以保证乱数的最小周期大于 $2^{40}$。*a* 和 *b* 可以分别是前一次调用结束时的累加值和生成的最后一个 32 比特乱数。在计算过程中，*a* 不断累加变化，*b* 也不断更新。

每次调用 ISAAC 加密算法的过程可以用如下伪代码来描述。

```
b=b+(++c);
for(i=0;i<256;i+=4)
{
    j=(i+128) mod 256;
    tmp=m[i];
    a=(a⊕(a<<13))+m[j];
    m[i]=y=m[(tmp>>2)&255]+a+b;
    r[i]=b=m[(y>>10)&255]+tmp;
    tmp=m[i+1];
    a=(a⊕(a>>6))+m[j+1];
    m[i+1]=y=m[(tmp>>2)&255]+a+b;
    r[i+1]=b=m[(y>>10)&255]+tmp;
    tmp=m[i+2];
    a=(a⊕(a<<2))+m[j+2];
    m[i+2]=y=m[(tmp>>2)&255]+a+b;
    r[i+2]=b=m[(y>>10)&255]+tmp;
    tmp=m[i+3];
    a=(a⊕(a>>16))+m[j+3];
    m[i+3]=y=m[(tmp>>2)&255]+a+b;
    r[i+3]=b=m[(y>>10)&255]+tmp;
}
```

上述 32 比特的字之间的加法均为模 $2^{32}$ 加。

ISAAC 加密算法中数组的大小和字长实际是可变的，但通常采用 32 比特，数组规模为 $2^8$=256 字。

## 3.4 FISH 加密算法

FISH（Fibonacci Shrinking Generator）是由 Siemens 公司的 Blocher 和 Dichtl 于 1993 年提出的一种快速软件序列加密算法。该算法以压缩发生器原理为基础，充分利用了一般微机处理器字长为 32 比特的特点，因而运行速度极快，这是该算法最突出的特点。在 Intel 486/33 上，用 C 语言编程，每秒能生成 15 MB 的数据。

FISH 加密算法由两个 Fibonacci 发生器构成，记为 $A$ 和 $S$，$A$ 和 $S$ 分别满足下面的递归关系：

$$A：a_i=a_{i-55}+a_{i-24} \pmod{2^{32}}$$

$$S：s_i=s_{i-52}+s_{i-29} \pmod{2^{32}}$$

$A$ 和 $S$ 的初态由输入密钥来预置。记 $s_i$ 的第 $j$ 比特为 $s_{ij}$，压缩规律如下：如果 $s_{i0}$=1，将 $a_i$ 添加到序列 $z_k$ 上，将 $s_i$ 添加到序列 $h_k$ 上。这样 $a_i$ 序列压缩成 $z_i$ 序列，$s_i$ 序列压缩成 $h_i$ 序列。

再对序列 $z_i$ 和 $h_i$ 进行如下变换：

$$c_{2i}=z_{2i} \oplus (h_{2i} \& h_{2i+1})$$

$$c_{2i+1}=z_{2i+1}$$

$$d_{2i}=h_{2i+1} \& (c_{2i} \oplus c_{2i+1})$$

$$r_{2i}=c_{2i} \oplus d_{2i}$$

$$r_{2i+1}=c_{2i+1} \oplus d_{2i}$$

实际上，上述后三个公式对 $c_{2i}$ 和 $c_{2i+1}$ 的一些比特进行了交换，即当 $h_{2i+1,j}$=1 时，$c_{2i+1,j}$ 与 $c_{2i,j}$ 互换。$r_i$ 数列即输出结果。

FISH 加密算法的输出序列具有很好的统计特性，但是用 $s_{i0}$ 作为控制位是它的一个弱点。由模数是 2 的幂次方的 Fibonacci 序列的最低比特构成了一个线性反馈移位寄存器序列，其特征多项式与 Fibonacci 序列的特征多项式一致。Ross Anderson 正是利用了这一弱点提出了攻击方法，而且在输出序列中，$r_{2i,0}=z_{2i+1,0}$，$r_{2i+1,0}=\bar{z}_{2i,0}$。

## 3.5 PIKE 加密算法

PIKE 加密算法是由 Ross Anderson 提出的，采用了一种改进的 Fibonacci 发生器。PIKE 加密算法采用了 A5 加密算法的基本设计思想，但纠正了 A5 加密算法中移位寄存器级数太低的不足，并改进了的 FISH 加密算法由最低比特作为控制位的弱点，具有极高的强度。

PIKE 加密算法主要由 3 个 Fibonacci 发生器构成，这 3 个 Fibonacci 发生器的反馈关系为：

$$a_i=a_{i-55}+a_{i-24}(\text{mod } 2^{32})$$
$$b_i=b_{i-57}+b_{i-7}(\text{mod } 2^{32})$$
$$c_i=c_{i-58}+c_{i-19}(\text{mod } 2^{32})$$

3 个 Fibonacci 发生器的初态由输入密钥来决定，这 3 个 Fibonacci 发生器的步进（停/走）由 $a_i$、$b_i$、$c_i$ 各自的进位比特来决定，若 $a_i$、$b_i$、$c_i$ 的进位比特都相同，则 3 个 Fibonacci 发生器都步进一步，否则进位比特相同的 2 个 Fibonacci 发生器步进一步，另外一个 Fibonacci 发生器停一步。进位比特的控制作用要延迟 8 拍。每次在 Fibonacci 发生器状态变化之后，检查 3 个进位比特，用 1 个符号表示出 3 个进位比特的控制步进情况，然后将这个符号写到一个移存寄存器中，该移位寄存器每次移位 4 比特。如果在某些处理器中，最高位是这个数的奇偶校验位，则可以用奇偶校验位作为控制比特。

输出值为：　　$r_i=a_i \oplus b_i \oplus c_i$

当每个 Fibonacci 发生器生成的序列长度达到 $2^{32}$ 时，3 个 Fibonacci 发生器的初态需要重新预置。

## 3.6　TWOPRIME 加密算法

TWOPRIME 加密算法是由芬兰的 Cunsheng Ding、ValtteriNiemi、Ari Renvall、Arto Salomaat 于 1997 年提出的，它是一个加法同步序列密码算法，以（$p$,$a$）计数器为乱源，经加乱、相加及代替后产生乱数序列，密钥长度为 128 比特。该算法具有以下几个特点：专为 32 位计算机设计，但又很容易修改成在 64 位计算机上快速运行；以字节为单位操作，软件实现速度快；结构特殊，安全度可控，能有效对付穷尽搜索攻击、线性复杂度攻击、反推攻击、相关攻击和仿射逼近攻击等。

128 比特的密钥按字节记成 $k_0k_1\cdots k_{15}$，然后按下式分成 4 个部分，分别记为 $K_0$、$K_1$、$K_2$、$K_3$。

$$K_0=(k_{11},k_{10},k_9,k_8)$$
$$K_1=(k_{15},k_{14},k_{13},k_{12})$$
$$K_2=(k_0,k_1,k_2,k_3)$$
$$K_3=(k_4,k_5,k_6,k_7)$$

TWOPRIME 加密算法编制框图如图 3.6.1 所示。

从编制框图可以看出，整个算法分为十层，前九层为加乱过程，最后一层为加乱过程。

第 1 层由 2 个计数器组成。每个计数器内有一个长度为 $\lceil \log_{2p} \rceil$ 的整数寄存器，设寄存器的初始值为 $K$（$0 \leqslant K \leqslant p-1$），则寄存器在 $i$ 时刻的值为：

$$r_i=(a*i+k)\bmod p,\qquad (a,p)=1 \tag{3-6-1}$$

从式（3-6-1）可以看出，计数器的步进为 $a$，周期为 $p$。将两个循环计数器分别记为$(p_0,a_0)$和$(p_1,a_1)$，$K_0$、$K_1$ 作为两个循环计数器的初始值，$p_0$ 和 $p_1$ 为两个不同的 32 位素数，$a_i$ 为不大于 $p_i-1$ 的常数，取 $p_0$ 和 $p_1$ 为两个最大的 32 位素数，$p_0$=4294967249，$p_1$=4294967291。$a_0$ 和 $a_1$ 的选取应遵循下列四条规则：

① $a_i>(p_i-1)/2$，$i$=0,1。

② $a_0\neq a_1$ 且$|a_0-a_1|$足够大。

③ $a_i$ 不能太接近 $p_i$。

④ 两者都为素数。

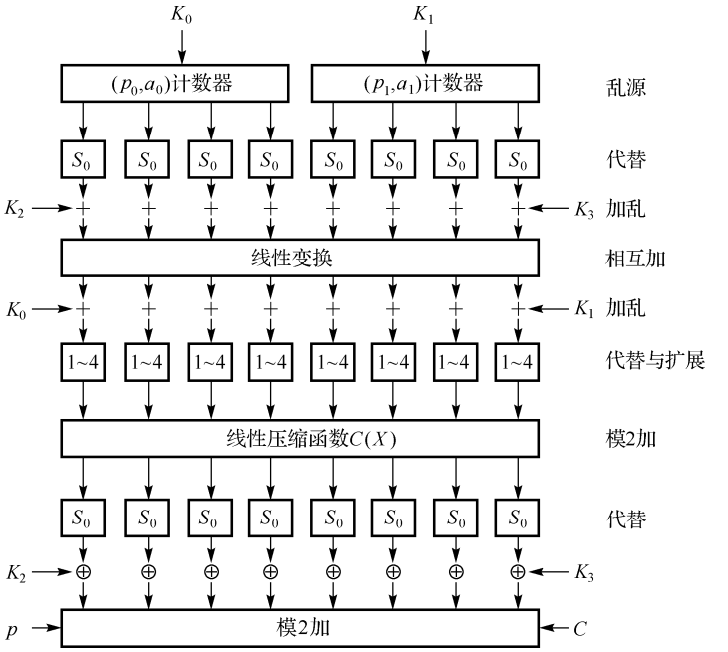



图 3.6.1　TWOPRIME 加密算法编制框图

按照以上规则，不妨取 $a_0$=2345986071，$a_1$=3124567807。

第 2 层使用 8 个相同的 $S$ 盒 $S_0$。$S_0$ 为加法剩余类环 $Z_{256}$ 上的一个非线性置换，$S_0(x)=(x^{255} \bmod 257) \bmod 256$，$x\in Z_{256}$。第 1 层的状态按字节作为 8 个 $S_0$ 盒的输入。

第 3 层是 8 个 $S_0$ 盒的输出与 $K_2$ 和 $K_3$ 的 8 个字节对应进行字节加，即模 256 加。

第 4 层是一个线性变换，主要用来进行密钥扩散。每个字节的输出是其他 7 个输入字节之和进行模 256 加的结果。

第 5 层类似于第 3 层，只是将第 3 层中的 $K_2$、$K_3$ 换成 $K_0$、$K_1$。

第 6 层是非线性变换层，用于数据代替与扩展，其输入为 8 字节，输出为 32 字节。每个包含符号 1～4 的方格表示每个输入字节都要经过 4 个 $S$ 盒 $S_1$、$S_2$、$S_3$、$S_4$ 变成 4 字节输出。4 个 $S$ 盒的定义为：

$$\begin{aligned}
&S_1(X)=(X^3 \bmod 257) \bmod 256\\
&S_2(X)=(X^{171} \bmod 257) \bmod 256\\
&S_3(X)=(45^X \bmod 257) \bmod 256\\
&S_4(X)=\begin{cases}(\log_{45} X \bmod 257) \bmod 256, & X\neq 0\\ 128, & X=0\end{cases}
\end{aligned} \tag{3-6-2}$$

第 7 层是一个线性压缩变换，它将 32 字节的输入变成 8 字节的输出。设 $X_0,X_1,\cdots,X_{31}$ 为 32 个输入字节，$Y_0,Y_1,\cdots,Y_7$ 为 8 个输出字节，则线性压缩函数 $C(X)$为：

$$\begin{aligned}
Y_0&=X_0+X_5+X_{10}+X_{15}+X_{16}+X_{22}+X_{24}+X_{30}\\
Y_1&=X_1+X_6+X_{11}+X_{12}+X_{17}+X_{23}+X_{25}+X_{31}\\
Y_2&=X_2+X_7+X_8+X_{13}+X_{18}+X_{20}+X_{26}+X_{28}\\
Y_3&=X_3+X_4+X_9+X_{14}+X_{19}+X_{21}+X_{27}+X_{29}\\
Y_4&=X_{16}+X_{21}+X_{26}+X_{31}+X_0+X_6+X_8+X_{14}\\
Y_5&=X_{17}+X_{22}+X_{27}+X_{28}+X_5+X_{11}+X_{13}+X_3\\
Y_6&=X_{18}+X_{23}+X_{24}+X_{29}+X_{10}+X_{12}+X_2+X_4\\
Y_7&=X_{19}+X_{20}+X_{25}+X_{30}+X_{15}+X_1+X_7+X_9
\end{aligned} \tag{3-6-3}$$

式（3-6-3）中的加法为模 256 加。

第 8 层同第 2 层一样。

第 9 层是 8 个 $S_0$ 盒的输出同 $K_2$ 和 $K_3$ 进行模 2 加后产生乱数序列。

至此加乱过程结束，乱数与明文进行模 2 加后产生密文。脱密过程和加密过程相同。

从上面的加乱过程可以看出，第 7 层的线性压缩函数 $C(X)$是最费时间的，因此 TWOPRIME 加密算法的提出者又给出了一个更简单的压缩函数 $C_1(X)$，即

$$\begin{aligned}
Y_0&=X_0+X_5+X_{10}+X_{15}\\
Y_1&=X_1+X_6+X_{11}+X_{12}\\
Y_2&=X_2+X_7+X_8+X_{13}\\
Y_3&=X_3+X_4+X_9+X_{14}\\
Y_4&=X_{16}+X_{21}+X_{26}+X_{31}\\
Y_5&=X_{17}+X_{22}+X_{27}+X_{28}\\
Y_6&=X_{18}+X_{23}+X_{24}+X_{29}\\
Y_7&=X_{19}+X_{20}+X_{25}+X_{30}
\end{aligned} \tag{3-6-4}$$

算法的其他部分保持不变，这样的加乱过程要比原来的过程快，这种更快速的版本称为TWOPRIME-1 加密算法。

TWOPRIME 加密算法还有一种变化形式，称为 ONEPRIME 加密算法，它是专为 64 位计算机设计的。二者的差别只是第 1 层不同，ONEPRIME 加密算法只有一个$(p,a)$计数器，$p$是最接近 $2^{64}-1$ 的素数，取 $p=18446744073709551557=2^{64}-59$，$a$ 是任一个接近 $3p/4$ 的素数。

## 3.7 密钥自动变化的字加密算法 WAKE

WAKE（Word Auto Key Ecryption）是英国剑桥大学计算机实验室的 Davie J. Wheeler 提出的加密算法，于 1993 年 12 月在剑桥算法研讨会上发表。WAKE 被认为是最快的加密算法之一，加密一个字（32 比特）大约需要 20 条指令。在 DEC 公司的 Alpha 芯片上每秒可处理约 100 MB 的数据。该算法以 32 比特字为单位进行运算，运算过程中密钥不断地自动变化。

WAKE 加密算法的逻辑框图如图 3.7.1 所示（图中箭头是指 32 比特字的流向）。

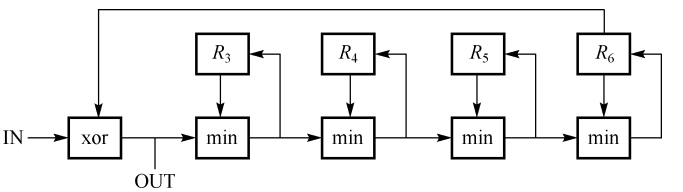



图 3.7.1　WAKE 加密算法的逻辑框图

首先将 4 个字（128 比特）的初始密钥输入寄存器 $R_3$、$R_4$、$R_5$、$R_6$，在加密过程中，密钥自动变化，变化后的新密钥仍然存放在 $R_3$、$R_4$、$R_5$、$R_6$。

设 $V[n]$为输入的明文字，经加密后输出密文字仍放在 $V[n]$（$n$=1,2,…）。对于每个 $n$ 进行以下变换：

$$R_1=V[n]$$

$$R_2=R_1 \text{ xor } R_6 \quad （\text{xor 表示逐位进行模 2 加}）$$

$$V[n]=R_2$$

$$R_3=M(R_3,R_2)$$

$$R_4=M(R_4,R_3)$$

$$R_5=M(R5,R_4)$$

$$R_6=M(R_6,R_5)$$

式中，$M(X,Y)=[(X+Y)]>>8 \text{ xor } t[(X+Y) \,\&\, 255]$；$t[\,]$为 256 个字的代替表（$t$ 表），该算法的 $t$ 表生成逻辑框图如图 3.7.2 所示，图中箭头表示 8 比特字节的流向。$M$ 函数中的 $t$ 表长度为 256

个字，记为 $t[0],t[1],\cdots,t[255]$，它由初始密钥 $K$（4 个字）按以下步骤生成。

（1）tt[0],…,tt[7]为置换表，令

```
tt[0]=0x726a8f3b
tt[1]=0xe69a3b5c
tt[2]=0xd3c71fe5
tt[3]=0xab3c73d2
tt[4]=0x4d3a8eb3
tt[5]=0x0396d6e8
tt[6]=0x3d4c2f7a
tt[7]=0x9ee27cf3
```

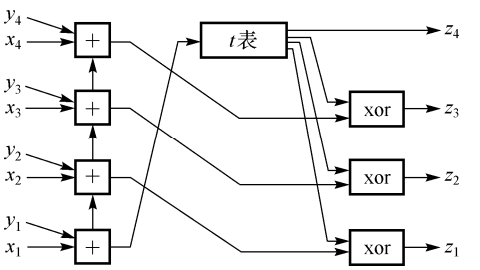



图 3.7.2　WAKE 加密算法的 $t$ 表生成逻辑框图

（2）令 $t[i]=K[i]$，$i$=0,1,2,3。对于 $p$=4～255，进行以下变换：

$$x=t[p-4]+t[p-1]$$

$$t[p]=(x>>3) \text{ xor } tt[y]$$

式中，$y$ 为 $x$ 低位 3 个比特的二进制数。

（3）将步骤（2）中产生的 $t[0]$～$t[255]$进行相互混乱。对于 $p$=0～22，进行以下变换：

$$t[p]=t[p]+t[p+89]$$

将 $t[33]$第 9 比特置 0 后送入 $x$，将 $t[59]$第 8 比特和最低比特置 1，以及第 9 比特置 0 后，送入 $z$，$x=x+z$。

对于 $p$=0～255，将 $t[p]$的前 8 比特置 0 后与 $x$ 进行模 2 加得到的值送入 $t[p]$。

（4）令 $t[256]=t[0]$，取 $x$ 的低 8 比特送入 $x$，对于 $p$=0～255，将 $t[p \text{ xor } x] \text{ xor } x$ 的值的低 8 比特送入 $x$，将 $t[x]$送入 $t[p]$，将 $t[p+1]$送入 $t[x]$。

加密程序函数 cypher( )和 $t$ 表生成程序函数 genkey()的伪代码如下。

```
cypher(v,n,k,r,t)
long v[ ],n,k[ ],r[ ],t[ ];
{
```

```
    long r1,r2,r3,r4,r5,r6,d,*e,m=0x00ffffff;
    r3=k[0];r4=k[1];r5=k[2];r6=k[3];
    if(n<0) d=-1;else d=1;
    e=v+n;
    while(v-n)
    {
        r1=*v;
        r2=r1^r6;
        *v=r2;
        v+=d;
        r3=r3+r2;
        r3=(r3>>8&m)^t[r3&255];
        r4=r4+r3;
        r4=(r4>>8&m)^t[r4&255];
        r5=r5+r4;
        r5=(r5>>8&m)^t[r5&255];
        r6=r6+r5;
        r6=(r6>>8&m)^t[r6&255];
    }
    r[0]=r3;r[1]=r[4];r[2]=r5;r[3]=r6;
}
genkey(t,k)
long t[ ],k[ ];
{
    long x,z,p;
    static long tt[10]={0x726a8f3b,0xe69a3b5c,0xd3c71fe5,
                    0xab3c73d2,0x4d3a8eb3,0x0396d6e8,0x9ee27cf3};
    for(p=0;p<4;p++)t[p]=k[p];          //copy k
    for(p=4;p<256;p++)
    {
        x=t[p-4]+t[p-1];                //fill t
        t[p]=x>>3 ^tt[x&7];
    }
    for(p=0;p<23;p++)t[p]+=t[p+89];
    x=t[33];z=t[59] | 0x01000001;
    z=z&0xff7fffff;
    for(p=0;p<256;p++)
    {
        x=(x&0xff7fffff)+z;
        t[p] =t[p] & 0x00ffffff ^ x;
    }
```

```
        t[256] =t[0];x&=255;
        for(p=0;p<256;p++)
        {
            t[p] =t[x] =(t[p^x]^x)&255];
            t[x] =t[p+1];
        }
    }
```

## 3.8　SEAL 加密算法

SEAL（Software Encryption Algorithm）加密算法是由美国 IBM 公司 P. Rogaway 和 D. Copper Smith 在 1993 年 12 月召开的第一次快速软件加密算法研讨会上提出的。这是一种面向软件实现的快速加密算法，平均每加密 1 字节只需约执行 5 条基本机器指令，比计算循环冗余码（Cyclic Redundancy Code，CRC）还要快。

SEAL 加密算法的乱数可以看成由 $\{0,1\}^{32}\rightarrow\{0,1\}^{L}$ 的伪随机函数 $\mathrm{SEAL}_K()$生成。对于给定的 160 比特密钥 $K$，SEAL 加密算法可将任意一个 32 比特的字 $n$ 映射到 $L$ 比特的伪随机数序列 $\mathrm{SEAL}_K(n)$，将 $\mathrm{SEAL}_K(n)$作为乱数，与 $L$ 比特的明文 $X$ 进行模 2 加后可得到密文。

乱数 $\mathrm{SEAL}_K(n)$不仅依赖于密钥 $K$，还依赖于明文 $X$ 的位置 $n$。这里 $n$ 是 32 比特的数，可以是序数，也可以是明文 $X$ 的物理地址。任选一个位置 $n\in\{0,1\}^{32}$，可以直接生成相应的 $L$ 比特乱数，或由此开始生成以 $L$ 比特为单位的乱数序列，这很适合某些应用。

SEAL 加密算法可分成三部分来描述：①由 160 比特密钥 $K$ 生成 $T$、$S$、$R$ 三张表，分别为 512 个字、256 个字和 $4t$ 个字，其中 $t=[(L-1)/8192]$；②初始化；③加乱。

（1）生成 $T$ 表、$S$ 表和 $R$ 表。$T$ 表、$S$ 表、$R$ 表是通过密钥 $K$ 和 SHA 计算得到的，SHA 是 NIST 在联邦信息处理标准 FIPS-181 号中规定的 Hash 标准。

定义 $G_K(j)$为以密钥 $K$ 的 5 个 32 比特字预置 SHA 中的 $H_0\sim H_4$，并通过 SHA 计算 $j$ 的 Hash 值得到的 5 个字。令：

$$T[i]=G_K(j)，0\leqslant i<512，j=0,1\cdots，\text{生成的前 512 个字}$$

$$S[i]=G_K(j)，0\leqslant i<256，j=0\text{x}1000,0\text{x}1001\cdots，\text{生成的前 256 个字}$$

$$R[i]=G_K(j)，0\leqslant i<4t，j=0\text{x}2000,0\text{x}2001\cdots，\text{生成的前 }4t\text{ 个字}$$

其中，$t=[(L-1)/8192]$，0x 表示十六进制数。

（2）初始化。初始化函数 $\mathrm{initialize}_K(n,m,A,B,C,D,n_1,n_2,n_3,n_4)$把 $n,m$ 映射到 $A,B,C,D,n_1,n_2,n_3,n_4$，即用 $R$ 表、$T$ 表和 $n$、$m$ 值预置 $A\sim D$ 和 $n_1\sim n_4$。

$$A\leftarrow n_1\oplus R[4m]$$

$$B \leftarrow (n_2 >>> 8) \oplus R[4m+1]$$
$$C \leftarrow (n_3 >>> 16) \oplus R[4m+2]$$
$$D \leftarrow (n_4 >>> 24) \oplus R[4m+3]$$

其中，“>>>”表示循环右移位。

对于 $j$=1～2，进行以下变换。

$$P \leftarrow A\&0x7fc,\quad B \leftarrow B+T[P/4],\quad A \leftarrow A>>>9$$
$$P \leftarrow B\&0x7fc,\quad C \leftarrow C+T[P/4],\quad B \leftarrow B>>>9$$
$$P \leftarrow C\&0x7fc,\quad D \leftarrow D+T[P/4],\quad C \leftarrow C>>>9$$
$$P \leftarrow D\&0x7fc,\quad A \leftarrow A+T[P/4],\quad D \leftarrow D>>>9$$

将 $D$、$B$、$A$、$C$ 映射到 $n_1$、$n_2$、$n_3$、$n_4$，即

$$(n_1,n_2,n_3,n_4) \leftarrow (D,B,A,C)$$
$$P \leftarrow A\&0x7fc,\quad B \leftarrow B+T[P/4],\quad A \leftarrow A>>>9$$
$$P \leftarrow B\&0x7fc,\quad C \leftarrow C+T[P/4],\quad B \leftarrow B>>>9$$
$$P \leftarrow C\&0x7fc,\quad D \leftarrow D+T[P/4],\quad C \leftarrow C>>>9$$
$$P \leftarrow D\&0x7fc,\quad A \leftarrow A+T[P/4],\quad D \leftarrow D>>>9$$

（3）由 32 比特生成 $L$ 比特随机数，即 $\mathrm{SEAL}_K(n)=y=(y_0,y_1,\cdots,y_{L-1})$，$\mathrm{SEAL}_K(n)$的实现如下。

开始令 $y$=0，对于 $m$=0～$t$–1，$t$=[($L$–1)/8192]，进行以下变换：

① $\mathrm{initialize}_K(n,m,A,B,C,D,n_1,n_2,n_3,n_4)$。

② 对于 $i$=1～64，进行以下变换：

$$P\leftarrow A\&0x7fc,\quad B\leftarrow B+T[P/4],\quad A\leftarrow A>>>9,\quad B\leftarrow B\oplus A$$
$$Q\leftarrow B\&0x7fc,\quad C\leftarrow C\oplus T[Q/4],\quad B\leftarrow B>>>9,\quad C\leftarrow C+B$$
$$P\leftarrow (P+C)\&0x7fc,\quad D\leftarrow D+T[P/4],\quad C\leftarrow C>>>9,\quad D\leftarrow D\oplus C$$
$$Q\leftarrow (Q+D)\&0x7fc,\quad A\leftarrow A+T[Q/4],\quad D\leftarrow D>>>9,\quad A\leftarrow A+D$$
$$P\leftarrow (P+A)\&0x7fc,\quad B\leftarrow B\oplus T[P/4],\quad A\leftarrow A>>>9$$
$$Q\leftarrow (Q+B)\&0x7fc,\quad C\leftarrow C+T[Q/4],\quad B\leftarrow B>>>9$$
$$P\leftarrow (P+C)\&0x7fc,\quad D\leftarrow D\oplus T[P/4],\quad C\leftarrow C>>>9$$
$$Q\leftarrow (Q+D)\&0x7fc,\quad A\leftarrow A+T[Q/4],\quad D\leftarrow D>>>9$$
$$y\leftarrow y \parallel B+S[4i-4] \parallel C\oplus S[4i-3] \parallel D+S[4i-2] \parallel A\oplus S[4i-1]$$

如果 $y$ 的比特数≥$L$，则结束，并返回$(y_0,y_1,\cdots,y_{L-1})$；如果 $i$ 是奇数，则$(A,C) \leftarrow (A+n_1,C+n_2)$；否则$(A,C) \leftarrow (A+n_3,C+n_4)$。

## 3.9　现代圆盘密码

圆盘密码（又称为线路轮密码）作为机械密码的两大类型之一，在 20 世纪 40 年代至 60 年代中期是许多国家军事和外交领域使用的主要密码。由于电子技术的发展，从 60 年代末，密码体制开始进入电子时代，以线性移位寄存器为乱源的序列密码逐步代替了机械密码，70 年代中又提出了以 DES 为代表的分组密码，密码体制全面进入了电子时代。然而圆盘密码的编码思想仍在影响着电子密码的发展，例如圆盘密码线路轮的不规则步进，发展演变为移位寄存器的不规则步进；圆盘密码的代替作业思想与分组密码的编制也有内在联系；特别是由圆盘密码和线性移位寄存器序列密码相结合产生的由电子方式实现的代替加乱体制，在电子密码中独树一帜。在某种意义上，电子密码就是以电子方式实现的圆盘密码，所以有人将其称为现代圆盘密码。

典型的圆盘密码是由 $n$ 个线路轮和轮子的运动规则组成的，或者说它是 $n$ 个 $m$ 元随机置换（即 $m$ 元非线性代替）以及每个时刻各置换的起点决定的密码变换。设 $n$ 个 $m$ 元置换为 $p_1,p_2,\cdots,p_n$，某时刻 $n$ 个置换的起点（即 $n$ 个线路轮的位置）为 $k_1,k_2,\cdots,k_n$。设输入明文为 $x$，输出密文为 $y$，则圆盘密码实现的明/密文变换为：

$$y=p_n(\cdots p_3(p_2(p_1(x+k_1)-k_1+k_2)-k_2+k_3)-k_3+\cdots)-k_n$$

令 $k_1'=k_1$，$k_i'=k_i-k_{i-1} \bmod m$（$i=2,3,\cdots,n$），$k_{n+1}'=-k_n \bmod m$，则

$$y=p_n(\cdots p_3(p_2(p_1(x+k_1')+k_2')+k_3')+\cdots)+k_{n+1}' \qquad (3\text{-}9\text{-}1)$$

明文 $x$ 先加 $k_1'$，再进行代替 $p_1$，然后加 $k_2'$，再进行代替 $p_2$，……，所以线路轮的位置实际上决定了在 $p_1,\cdots,p_n$ 各代替之间进行一次模 $m$ 的加法运算，因此 $n$ 个线路轮的圆盘密码可以看成一种 $n$ 层代替加乱体制。不过圆盘密码中的加乱是由线路轮的运动规则决定的。若以电子元件来实现非线性代替 $p_i$，则由线性移位寄存器来提供 $k_i$ 值，这就是我们所说的代替加乱体制，即所谓的现代圆盘密码。

现代圆盘密码可以有多种变型：加乱数 $k_i$ 可以采用模 $m$ 加，也可以采用模 2 加；乱数 $k_i$ 可以由线性移位寄存器来提供，也可以由其他因素来决定，例如，由前面变换的中间结果累加得到（见后面介绍的第二种现代圆盘密码）；$n$ 个代替 $p_1,\cdots,p_n$ 可以像直路圆盘那样各作用一次进行复合，也可以像回路圆盘那样再反向复合一次。下面介绍几种在公开文献中提到的现代圆盘密码。

（1）英国剑桥大学的 Ross Anderson 在 1993 年 12 月召开的第一次快速软件加密算法研讨会上给出的现代圆盘密码是由 3 个 256 元随机置换（$p_1$、$p_2$、$p_3$）和一个线性移位寄存器构成的，即 $n=3$，$m=256$。明/密文都是{0,1,…,255}中的元素。明/密文变换为：

$$y=p_3\{p_2[p_1(x+k_1)+k_2]+k_3\}+k_4 \tag{3-9-2}$$

式中，$k_i$ 是由线性移位寄存器中选取的 3 个 8 比特数决定的（按原文中的表述，若线性移位寄存器提供的 3 字节为 $r_1$、$r_2$、$r_3$，则 $k_1=r_1$，$k_2=r_2-r_1 \bmod 256$，$k_3=r_3-r_2 \bmod 256$，$k_4=-r_3 \bmod 256$，实际上可以定义 $k_1$～$k_4$ 为直接取自线性移位寄存器的 4 个字节）。这里的加乱是模 256 加。如果将 0～255 中的每个元素表示成由 8 比特组成的一个字节，并且加乱采用模 2 加，则明/密文变换为：

$$y=p_3\{p_2[p_1(x\oplus k_1)\oplus k_2]\oplus k_3\}\oplus k_4 \tag{3-9-3}$$

无论模 256 加还是模 2 加，都是一种特殊的置换，都可以将明/密文变换以置换的乘积形式来表示。例如，引入循环置换：

$$\sigma=\begin{bmatrix}0 & 1 & 2 & \cdots & m-2 & m-1\\ 1 & 2 & 3 & \cdots & m-1 & 0\end{bmatrix}$$

则式（3-9-2）可以表示成

$$y=\sigma^{k_4}p_3\sigma^{k_3}p_2\sigma^{k_2}p_1\sigma^{k_1}(x) \tag{3-9-4}$$

还可类似地引入模 2 加置换。

（2）德国卡塞鲁大学的 Peer Wichmann 在 1989 年欧州密码年会上发表了一篇对现代圆盘密码进行分析的文章，提到在某些操作系统中使用了现代圆盘密码来加密文件，该论文中给出的现代圆盘密码的逻辑框图如图 3.9.1 所示。

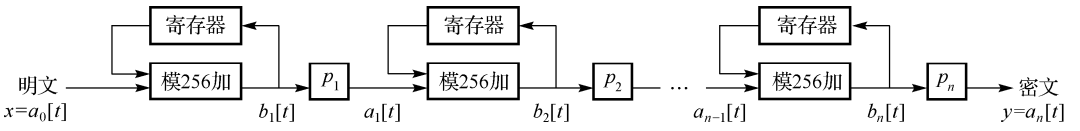



图 3.9.1　现代圆盘密码的逻辑框图

输入的明文和输出的密文均为 8 比特，$p_1$～$p_n$ 均为 256 元随机置换。第 $t$ 时刻输入明文 $x$ 记为 $a_0[t]$，第一次加乱后记为 $b_1[t]$，经 $p_1$ 代替后记为 $a_1[t]$，再经第二次加乱后记为 $b_2[t]$，经 $p_2$ 代替后记为 $a_2[t]\cdots$，最后经 $p_n$ 代替，输出 $a_n[t]$，即密文 $y$。第 $t$ 时刻第 $i$ 层的加乱是模 256 加 $b_i[t-1]$，即加上前一时刻该层的加乱结果。$a_0[t]$是 $t$ 时刻输入的明文，$a_n[t]$是 $t$ 时刻输出的密文，上述过程用公式表达如下：

$$b_i[0]=0,\quad i=1,2,\cdots,n$$

$$b_i[t]=a_{i-1}[t]+b_i[t-1] \bmod 256$$

$$a_i[t]=p_i(b_i[t]),\quad i=1,2,\cdots,n;\ t=1,2,3\cdots$$

显然，加乱受到了明文的控制。

（3）美国 AT&T Bell 实验室的 J. A. Reeds 和 P. J. Weinberger 在 1984 年 Bell 实验室的技术杂志上发表了“The UNIX System: File Security and the UNIX System Crypt Command”（文件安全和 UNIX 系统的 Crypt 命令），该论文介绍了文件加密命令 Crypt，该命令类似于简单的回路圆盘密码。Crypt 命令将待加密的明文每 256 个字符分成一组，将第 $j$ 组的第 $i$ 个字符记为 $x_{ij}$，加密后的相应密文记为 $y_{ij}$。明/密文均为 8 比特组成的字符。明/密文变换为：

$$y_{ij} = p_1^{-1}[p_2\{p_1(x_{ij}+i)+j\}-j]-i \qquad (3\text{-}9\text{-}5)$$

式中，$p_1$ 和 $p_2$ 均为 256 元置换，且 $p_2$ 由 128 对互代构成，具有自逆的特点。式（3-9-5）中的加乱用循环置换表示，即

$$y_{ij} = \sigma^{-i}p_1^{-1}\sigma^{-j}p_2\sigma^{j}p_1\sigma^{i}(x) = (\sigma^{j}p_1\sigma^{i})^{-1}p_2\sigma^{j}p_1\sigma^{i}(x)$$

明/密文之间的代替关系与 $p_2$ 共轭，因此也具有互代的特点。

以上介绍了在文献中见到的三种现代圆盘密码，目的是通过公开文献中见到的编密手法来开阔思路。

## 3.10　HKM/HFX 传真加密系统

1994 年，HKM/HFX 传真加密系统被提交国际电信联盟（International Telecommunication Unit，ITU），建议用于安全等级为商业级的传真消息的验证和加密。HKM/HFX 在设计上考虑了对密码产品的出口限制，会晤密钥仅为 12 位的十进制数（40 比特），密码强度比较弱。

HKM 和 HFX 40 是传真加密系统 HKM/HFX 所用的两个加密算法，前者用于加密密钥，后者用于加密数据。它们都用组合乘同余算法生成乱数的序列密码，由乱数和明文相加得到密文。HKM 加密算法的明乱（乱数和明文）相加是模 10 加，HFX40 加密算法的明乱相加是模 2 加，所以加密后的密钥是十进制数，而加密后的传真消息（数据）是二进制比特序列。在本节中，$X$ 位均指十进制的位数，对于二进制位均明确说明。

### 1．密钥

HKM/HFX 传真加密系统有下列密钥：

（1）$CRYP_A$：16 比特，传真机 A（简称 A 机）的主密钥，用于生成 A 机与其他通信各方的传真机加密通信的主密钥 $MP_{A*}$。

（2）$MP_{AB}$：16 比特，加密 A 机发送给 B 机消息的主密钥，用于加密会晤密钥。

（3）$OT_{AB}$：6 比特，A 机与 B 机共享的一次性密钥，是通过某种安全途径预先分发的，用于加密 $MP_{AB}$ 和 $MP_{BA}$。

（4）SK：12 比特，会晤密钥，在加密传输消息之前由发方传真机（A 机）生成，用于加密传真消息，并以 $MP_{AB}$ 和随机数 RS 为密钥，加密传送给收方传真机（B 机）。

（5）RS：4 比特，随机数，在加密传输消息前由发方传真机随机选择，作为附加密钥与 $MP_{AB}$ 一起作为密钥加密 SK。

与密钥有关的因素还有 $F_A$ 和 $ID_A$，它们与 A 机的 $CRYP_A$ 和 B 机的 $F_B$ 一起生成 $MP_{AB}$，当 A 机作为收方时，它们又与 $CRYP_A$ 以及发方的 $F_B$ 一起加密对方发来的 $MP_{AB}$。

（1）$F_A$：6 比特，A 机号码的后 6 比特。

（2）$ID_A$：48 比特，A 机的识别符。

再引入两个符号 $TK_{AB}$ 和 $RCS_{AB}$，它们是加密后的 $MP_{AB}$。

（1）$TK_{AB}$：16 比特，A 机以 $OT_{AB}$ 为密钥加密 $MP_{AB}$ 的结果。

（2）$RCS_{AB}$：16 比特，B 机以 $F_A$、$F_B$、$ID_B$、$CRYP_B$ 为密钥加密 $MP_{AB}$ 的结果。

### 2．加密通信流程

HKM/HFX 传真加密系统有两种工作方式，即注册方式和自动工作方式，分别完成两个不同阶段的任务。

（1）注册方式（Registration Mode）。A 机和 B 机首次进行通信时，先以注册方式按图 3.10.1 所示的流程进行通信。

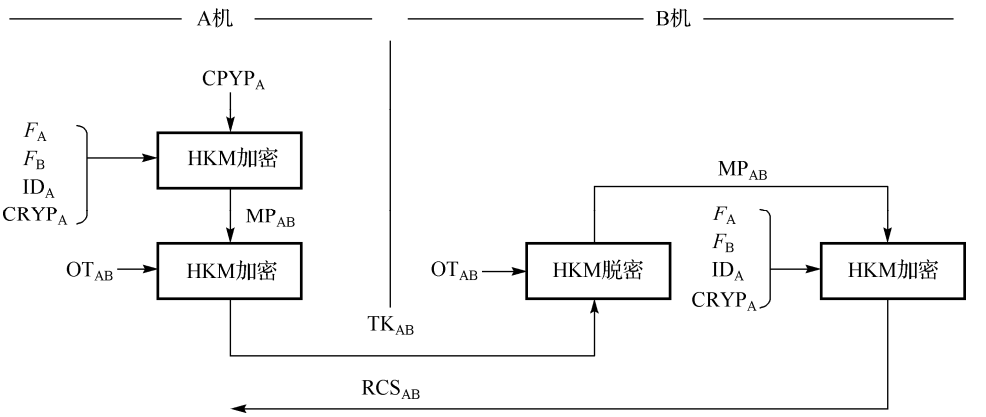



图 3.10.1　HKM/HFX 传真加密系统以注册方式进行通信的流程

以注册阶段方式进行通信的流程可分为 7 步：

① A 机和 B 机在以注册方式工作之前，先以某种安全的方式获得共同的 6 比特的一次性密钥 $OT_{AB}$。

② A 机以 $F_A$、$F_B$ 为密钥，用 HKM 算法加密 $CRYP_A$，生成 $MP_{AB}$，$MP_{AB}$ 是 A 机向 B

机发消息的单向主密钥（单向是指 $MP_{BA}$ 与 $MP_{AB}$ 不同），在双方通信期间是不变的。

③ A 机以 $OT_{AB}$ 为密钥，用 HKM 算法加密 $MP_{AB}$，得到 $TK_{AB}$。

④ A 机把 $TK_{AB}$ 发送给 B 机。

⑤ B 机用 $OT_{AB}$ 脱密 $TK_{AB}$，得到 $MP_{AB}$。

⑥ B 机用 $CRYP_B$、$ID_B$ 和双方号码的后 6 位数 $F_A$、$F_B$ 为密钥，按 HKM 算法加密 $MP_{AB}$，得到 $RCS_{AB}$。

⑦ B 机把 $RCS_{AB}$ 发送给 A 机，A 机将 $RCS_{AB}$ 与 B 机号码一起存储，待以后 A 机向 B 机发送消息时加密用。

（2）自动工作方式（Automatic Mode）。在以注册方式完成通信后，A 机可以在任何时间以自动工作方式向 B 机发送加密的消息。自动工作方式的通信流程如图 3.10.2 所示。

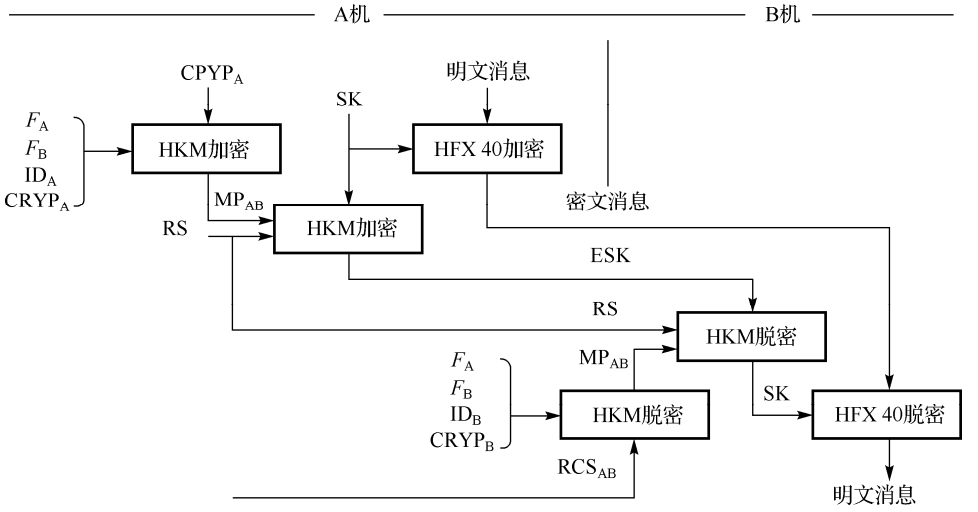



图 3.10.2　自动工作方式的通信流程

A 机向 B 机发送加密消息的通信流程可分为 5 步：

① A 机重新生成 $MP_{AB}$。

② A 机随机生成 12 比特会晤密钥 SK 和 4 比特随机数 RS，并以 $MP_{AB}$ 和 RS 为密钥，按 HKM 算法加密 SK，得到加密后的 SK，记为 ESK。

③ A 机把 $RCS_{AB}$、ESK 和 RS 发送给 B 机。

④ B 机脱密 $RCS_{AB}$，得到 $MP_{AB}$，再用 $MP_{AB}$ 和 RS 脱密 ESK，得到与 A 机共同的会晤密钥 SK。

⑤ A 机以 SK 为密钥，按 HFX 40 加密算法加密明文消息后并发送给 B 机，B 机用双方共有的会晤密钥 SK 脱密，得到明文消息。

### 3．HKM/HFX 概述

每台传真机（A 机）有其唯一的 48 位识别符 $ID_A$ 和一个 16 比特主密钥 $CRYP_A$，以及一个号码 $F_A$（只用后 6 比特），它们参与密钥的生成。此外，每台传真机中都装有 19 个相同的素数 $p_0,p_1,\cdots,p_{18}$，具体数值分别为 32603、32507、32183、32003、31847、31607、31583、31547、31259、31139、30803、30539、30467、30347、30323、30203、29879、29759、29663。

（1）HKM 加密算法。HKM 加密算法用于加密十进制的密钥，加密对象有三种：

① 以双方传真机号码（$F_A$ 和 $F_B$）、本机 $ID_A$ 和 $CRYP_A$ 为密钥，加密 $CRYP_A$ 生成 $MP_{AB}$ 或加密 $MP_{AB}$ 生成 $RCS_{AB}$，这时密钥为 76 比特。

② 以 $OT_{AB}$ 为密钥加密 $MP_{AB}$，这时密钥仅 6 比特。

③ 以 $MP_{AB}$ 为密钥加密 SK，这时密钥为 16 比特。

HKM 加密算法由下列 3 步组成：

① 由输入密钥（位数不等）形成 18 个整数，即 $P[0]$～$P[8]$和 $B[0]$～$B[8]$，作为生成乱数序列的初态和参数。

② 由 $P[i]$和 $B[i]$生成十进加乱数序列 $K[i],i=0,1\cdots$。

$$K[i]=\sum_{j=0}^{8}(P[j]\times B[j]^{i+1} \bmod P_j) \bmod 10$$

③ 明文和乱数进行模 10 加即可得到密文。

（2）HFX 40 加密算法。HFX 40 加密算法用于加密消息（数据），以 SK 为密钥进行加密，由下列 3 步组成：

① 由 12 比特的 SK 选择乘同余算法的模数 $p_0$、$p_1$、$p_2$，形成整数 $P[0]$～$P[2]$和 $B[0]$～$B[2]$，作为乘同余序列的初态和乘子。

② 由 $P[i]$和 $B[j]$生成二元乱数序列，即

$$K_i=[(P[0]\times B[0]^{(i \bmod 1021)+1} \bmod p_0)+(P[1]\times B[1]^{(i \bmod 1019)+1} \bmod p_1)+(P[2]\times B[2]^{(i \bmod 1013)+1} \bmod p_2)] \bmod 2$$

③ 明文与乱数进行模 2 加，得到密文。

乱数 $K_i$ 是 3 个乘同余序列的模 2 加，其周期为 1021×1019×1013=1053924187。

### 4．由密钥形成加乱初态和参数的具体步骤

（1）由 $F_A$、$F_B$、$ID_A$、$CRYP_A$ 生成 $P[0]$～$P[8]$和 $B[0]$～$B[8]$。$F_A$ 的高 3 比特和低 3 比特分别记为 $F_A[0]$和 $F_A[1]$，$F_B$ 类似。

$ID_A$ 共 48 比特，自前往后分成 13 份，分别记为 ID[0]～ID[12]，其中 ID[7]和 ID[8]为 2 比特，其他均为 4 比特。

$CPYP_A$ 共 16 比特，分成 5 份，记为 $c[0]$～$c[4]$，其中 $c[0]$～$c[2]$为 4 比特，$c[3]$和 $c[4]$各 2 比特，由 $F_A$、$F_B$、$ID_A$、$ID_B$、$CRYP_A$、$CRYP_B$ 的位数形成 $P[i]$、$B[j]$。

$$P[0]=ID[0]+F_A[0]$$
$$P[1]=ID[1]+F_A[1]+101$$
$$P[2]=ID[2]+F_B[0]+202$$
$$P[3]=ID[3]+F_B[1]+303$$
$$P[i]=ID[i]+i\times 101,\quad i=4,\cdots,8$$
$$B[0]=ID[9]$$
$$B[1]=ID[10]+79$$
$$B[2]=ID[11]+2\times 79$$
$$B[3]=ID[12]+3\times 79$$
$$B[i]=c[i-4]+i\times 79,\quad i=4,\cdots,8$$

由 $P[i]$和 $B[j]$生成 $K[0]$～$K[15]$，加密 $CRYP_A$，生成 $MP_{AB}$。

（2）由 $OT_{AB}$ 生成 $P[0]$～$P[8]$和 $B[0]$～$B[8]$。$OT_{AB}$ 共 6 比特，重复 10 次，再加上 $OT_{AB}$ 的前 4 比特，共 64 比特。从前往后划分成 18 份，记为 $T[0]$～$T[17]$。除了 $T[7]$、$T[8]$和 $T[16]$、$T[17]$为 2 比特，其他均为 4 比特。

$$P[i]=T[i]+i\times 101,\quad i=0,1,\cdots,8$$
$$B[i]=T[i-9]+i\times 79,\quad i=0,1,\cdots,8$$

由 $P[i]$和 $B[j]$生成乱数 $K[0]$～$K[15]$，加密 $MP_{AB}$，得到 $TK_{AB}$。

（3）由 $F_A$、$F_B$、$ID_B$、$CRYP_B$ 生成 $P[0]$～$P[8]$和 $B[0]$～$B[8]$。与步骤（1）类似地生成 $P[i]$和 $B[j]$（只是将 $ID_A$ 变成 $ID_B$，其他均不变），再由 $P[i]$和 $B[j]$生成乱数 $K[0]$～$K[15]$，加密 $MP_{AB}$，得到 $RCS_{AB}$。

（4）由 $MP_{AB}$ 生成 $P[0]$～$P[8]$和 $B[0]$～$B[8]$。$MP_{AB}$ 为 16 比特，重复 4 次共 64 比特，划分成 18 份，记为 $M[0]$～$M[17]$，除了 $M[7]$、$M[8]$和 $M[16]$、$M[17]$为 2 比特，其他均为 4 比特。将 4 比特随机数分成 2 份，各 2 比特，记为 $R[0]$和 $R[1]$。

$$P[0]=M[0]+R[0]$$
$$P[1]=M[1]+R[1]+101$$
$$P[i]=M[i]+i\times 101,\quad i=2,3,\cdots,8$$
$$B[i]=M[i+9]+i\times 79,\quad i=0,1,\cdots,8$$

由 $P[i]$和 $B[j]$生成乱数 $K[0]$～$K[11]$，加密 SK 发送给 B 机。

（5）由会晤密钥 SK 生成 $P[0]$～$P[2]$和 $B[0]$～$B[2]$。SK 为 12 比特，分成 6 份，各 2 比特，再分别加上 1024 即可得到 $P[0]$～$P[2]$和 $B[0]$～$B[2]$。由 SK 决定从传真机中装有的 19 个素数

中选择模数 $p_0$、$p_1$ 和 $p_2$，将 SK 的前 3 比特记为 SK(0,1,2)，设 SK(0,1,2) mod 19 =$t$，则选择 $p_t$ 作为 $p_0$。类似地通过 SK(3,4,5)和 SK(6,7,8)分别决定 $p_1$、$p_2$。

## 3.11 PKZIP 加密算法

PKZIP 加密算法广泛用于计算机存档文件的压缩编码程序，它是一种序列加密算法，供用户在选择加密时使用。该加密算法在输入明文数据的控制下，以字节为单位产生乱数，与明文结合产生密文，它是有别于通常序列密码加乱模式的又一种序列密码，该加密算法以口令作为密钥，密钥长度的字节数可变，其设计者为 Roger Schlafly。

输入的明文先进行压缩，在压缩后的明文前加上文件头，文件头包含下列内容：文件名+字节，压缩方法+字节，文件是否加密，CRC-32 值，原文件和压缩文件的大小，以及其他辅助信息。从几种压缩算法中选使压缩文件最短的方法对文件进行压缩。如果压缩算法不能使文件缩短，则文件不压缩，以原形存放。

如果要进行加密，则在压缩文件之前、文件头之后插入 12 字节的密文头，记为 $P_1$～$P_{12}$。经压缩处理的明文记为 $P_{13}$～$P_n$。文件头是不加密的，加密从密文头开始。$P_1$～$P_{12}$ 用于随机化，另外，其中某些 $P_i$ 依赖于文件头，在脱密时可以用来校验脱密密钥是否正确。

PKZIP 加密算法的加乱主要用了 3 个 32 比特的寄存器（无符号长整数），记为 $R_1$～$R_3$。由密钥对 $R_i$ 进行初始化，由输入的明文不断改变 $R_1$～$R_3$ 的状态。将 $j$ 时刻 $R_i$ 的状态记为 $R_i(j)$，$i$=1,2,3，$j$=1,2,3…。由 $R_i(j)$低位 2 个字节生成 1 字节的乱数，记为 $K_j$，$j$=1,2…。

PKZIP 序列密码的主要功能模块是 update-r(unsigned short char)。下面先介绍 update-r()要用到的循环冗余码函数 crc32(unsigned long char)。crc32()先将一个 32 比特的无符号长整数与 8 比特的一个字符进行逐比特模 2 加，然后将模 2 加的结果逐一循环右移 8 次，每次右移 1 比特，若移出去的比特为 1，则与常数 $C$=0xEDB88320（十六进制数）进行模 2 加。用 C 语言描述的伪代码如下：

```
crc32 (temp,i)
unsigned long temp;
char i;
{
    register int j;
    temp=temp⊕i;  /*表示 temp 和 i 进行逐比特模 2 加*/
    for (j=0; j<8; j++)
    if odd(temp) temp=(temp>>1)^0xEDB88320;
    else temp=temp>>1
    return (temp)
}
```

update-r(unsigned short char)功能实现的伪代码如下：

```
unsigned short temp;              /*定义 temp 为 16 比特的无符号短整数*/
R1(j+1)=crc32(R1(j),char);
R2(j+1)=(R2(j)+LSB(R1(j+1)))*134775813+1 mod 2^32;
R3(j+1)=crc32(R3(j),MSB(R2(j+1)));
temp=R3(j+1)&0xffff|3;            /*取 R3 的低 2 字节,并将最低 2 比特置 1*/
Kj+1=LSB((temp*(temp⊕1))>>8);     /*由 R3 生成 1 字节乱数 K*/
```

其中，LSB 和 MSB 分别表示最低字节和最高字节，“|”表示按比特进行或运算。

设密钥 key 为 $t$ 字节，记为 key[1]～key[$t$]，初始化过程如下：

$$R_1(1-t)=0x12345678$$

$$R_2(1-t)=0x23456789$$

$$R_3(1-t)=0x34567890$$

对于 $j$=1～$t$，执行 update-r(key[$j$])。经初始化后的 $R_i$ 即 $R_i(1)$，$i$=1,2,3，$K$ 值即 $K_1$。

对 $P_1$～$P_n$（其中 $P_1$～$P_{12}$ 为密文头）加密过程如下：

对于 $j$=1～$n$，进行以下变换：

$$C_j=P_j \oplus K_j$$

$$\text{Update-r}(P_j)$$

即用明文不断更新 $R_i$，不断生成乱数字节 $K_j$，用于加密明文。在脱密时，同样初始化 $R_i$，得到 $R_i(1)$和 $K$。

对于 $j$=1～$n$，进行以下变换：

$$P_i=K_i \oplus C_i$$

$$\text{update–r}(P_i)$$

得到 $P_1$～$P_n$，去掉密文头 $P_1$～$P_{12}$，即可得到压缩处理后的明文。

加密算法 PKZIP 在已知 13～40 个明文字节(压缩处理后的)的条件下，可以还原出 $R_i(0)$。

## 3.12　CA 自动机在序列密码乱源中的应用

细胞结构自动机（Cellular Automata，CA）是移位寄存器（或级间模 2 加移存器）的推广，可看成一种特殊的自动机。近十几年来，不断有关于 CA 在密码学中的应用以及对 CA 进行密码分析的论文发表。本节主要介绍模 2 域 GF(2)上的 CA（已有文献将 CA 的讨论推广到了 GF($q$)上）在序列密码乱源中的应用。在本节的讨论中，如果不加以特别说明，都是指模 2 域 GF(2)上的 CA。

### 3.12.1　基本概念和定义

为了阅读方便，本节给出的概念尽量与在移位寄存器中所用的概念一致，所以会与一些

文献中的概念有差别。例如，移位寄存器中的一级（Stage）在 CA 中称为一个单元或一个细胞（Cell）。

### 1．CA 的基本定义

有些文献资料将 CA 定义为多个单元组成的阵列（Array），每个单元处于所有可能的状态在 GF(2)上只有“0”“1”两种可能，在每个时钟周期，每个单元的状态按某种规则发生变化，变化后的状态是当前自身状态和相邻状态的函数。

用移位寄存器的相关概念来说，一维 CA 是一种特殊的多回头反馈移位寄存器，其每级的下一状态是其本身和相邻状态的函数。最常用的 CA，其下一状态是由自身和相邻的两级状态决定的，这种 CA 称为 3-邻接 CA（3-Neighborhood CA）。把 $t$ 时刻 CA 的状态记为 $S(t)$，$n$ 级 CA 的状态可表示为：

$$S(t)=[S_1(t),\cdots,S_n(t)]$$

则 3-邻接 CA 下一状态 $S(t+1)$中各分量为：

$$S_i(t+1)=f_i[S_{i-1}(t),S_i(t),S_{i+1}(t)] \tag{3-12-1}$$

式中，$f_i$ 为反馈函数。类似地可以定义 $k$-邻接 CA。也有文献将满足式（3-12-1）递归关系的 CA 称为 1-邻接 CA，将 $k$-邻接 CA 定义为：

$$S_i(t+1)=f_i[S_{i-k}(t),S_{i-k+1}(t),\cdots,S_i(t),\cdots,S_{i+k}(t)]$$

目前讨论较多且比较实用的是式（3-12-1）给出的 3-邻接 CA，下面所说的 CA 均指 3-邻接 CA。

### 2．线性 CA 与非线性 CA

若 CA 的反馈函数仅依赖于模 2 加，则称为加法 CA（Additive CA）。加法 CA 是线性的或仿射的。对于 3-邻接加法 CA，可以有 $2\times2^3=16$ 种不同的反馈函数，即：

$$\{f(S_{i-1},S_i,S_{i+1})=C_0\times1\oplus C_1\times S_{i-1}\oplus C_2\times S_i\oplus C_3\times S_{i+1}|C_j\in\{0,1\},\quad j=0,1,2,3\}$$

我们将加法 CA 称为线性 CA。线性 CA 可以用状态转移矩阵表示，即用

$$\boldsymbol{A}=\begin{bmatrix} d_1 & b_1 & 0 & \cdots & 0 & c_1 \\ c_2 & d_2 & b_2 & \cdots & 0 & 0 \\ 0 & c_3 & d_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-1} & b_{n-1} \\ b_n & 0 & 0 & \cdots & c_n & d_n \end{bmatrix} \tag{3-12-2}$$

来表示。有的文献讨论了非线性的 CA，其反馈函数为：

$$f_i(S_{i-1},S_i,S_{i+1})=S_{i+1}\oplus(S_i\vee S_{i+1})$$

3-邻接加法 CA 仅占所有可能的 3-邻接 CA 的 1/16。

### 3．循环 CA 和非循环 CA

当 $S_i(t+1)$只依赖于 $S_1(t)$和 $S_2(t)$，且 $S_n(t+1)$只依赖 $S_{n-1}(t)$和 $S_n(t)$时，称为非循环 CA。对于线性 CA，若式（3-12-2）中的 $c_n=b_n=0$，则为非循环 CA；反之，可将 CA 的各级看成一个圈，头尾相接，头尾两级和中间各级一样依赖于自身和相邻的两级，即：

$$S_i(t+1)=f_i[S_n(t),S_1(t),S_2(t)]$$

$$S_n(t+1)=f_n[S_{n-1}(t),S_n(t),S_1(t)]$$

则称为循环 CA。有文献中将这两种情况称为不同的边界条件，将前者称为空边界（Null Boundary），后者称为循环边界（Periodic Boundary 或 Cyclic Boundary）。

### 4．单一 CA 和混合 CA

当 CA 的各级反馈函数均相同时，即 $f_1=f_2=\cdots=f_n$，称为单一 CA；否则称为混合 CA。

### 5．程序可控 CA

若 CA 在控制信息的作用下，可对不同的时钟周期采用不同的状态转移变换，则称为程序可控 CA（Programmable Cellular Automata，PCA）。

## 3.12.2　CA 乱数发生器

CA 有多方面的应用，除了加密，还可用于测试模式的生成（Test Pattern Generation）和伪随机数生成（Pseudorandom Number Generation）等。在密码学中，CA 主要用来替代序列密码中的线性反馈移位寄存器，也有人用它来构建公钥密码或分组密码的基元，但在这方面的讨论比较少。本节仅介绍 CA 在序列密码中的应用。

将 CA 用于加密是由美国 Princeton 高级研究所的 Stephen Wolfram 首先提出的，他在 1985 年美洲密码年会上发表了论文 *Cryptography with Cellular Automata*，提出了一种简单的一维非线性 CA。此后又有人提出了各种线性 CA，用来代替线性反馈移位寄存器（LFSR），并把序列密码中的各种编密技术，如非线性组合函数、不规则步进（停/走）等引入 CA 乱数发生器。现将有关内容归纳简介如下。

### 1．非线性反馈的单一 CA

根据 Stenphen Wolfram 的研究，有两种非线性反馈单一 CA 的伪随机特性比较好，它们的递归关系式为：

$$S_i(t+1)=S_{i-1}(t)\oplus[S_i(t)\vee S_{i+1}(t)] \qquad (3\text{-}12\text{-}3)$$

和

$$S_i(t+1)=S_{i-1}(t)\oplus[S_i(t)\vee \overline{S_{i+1}(t)}] \qquad (3\text{-}12\text{-}4)$$

这等价于

$$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)\oplus S_{i+1}(t)\oplus S_i(t)\oplus S_{i+1}(t)$$

和

$$S_i(t+1)=S_{i-1}(t)\oplus S_{i+1}(t)\oplus S_i(t)S_{i+1}(t)\oplus 1$$

Stenphen Wolfram 在 1985 年美洲密码年会上对式（3-12-3）定义的 CA 进行了讨论，指出当 $n$ 比较大时，大多数的状态都进入最长的圈，最长的圈上约有 $2^{0.61n}$ 个状态。Stenphen Wolfram 在论文中还提到，这方面的许多工作是在为 Thinking Machines Corporation 做咨询时进行的，表明这方面的工作是有实际背景的。

在 1991 年欧洲密码年会上，瑞士的 O. Staffelbach 和 W. Meier 讨论了这类 CA 的综合算法。他们给出了对某个 $i$，由 $S_i(t)$求初态的方法和计算量（$t$=1,2…），并指出当 $n$=300～500 时，在 PC 上可以很快地由 $S_i(t)$求出初态；当 $n$=1000 时，使用大的并行机进行计算，也可求出初态。

### 2．作为级间模 2 加线性反馈移位寄存器推广的线性 CA

3-邻接线性 CA 共有 16 种反馈模式，在超大规模集成电路（VLSI）设计中的编号为规则（rule）0、15、51、60、85、90、102、105、150、153、165、170、195、204、240 和 255，其反馈关系分别为：

rule　0：$S_i(t+1)=0$

rule　15：$S_i(t+1)=S_{i-1}(t)\oplus 1$

rule　51：$S_i(t+1)=S_i(t)\oplus 1$

rule　60：$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)$

rule　85：$S_i(t+1)=S_{i+1}(t)\oplus 1$

rule　90：$S_i(t+1)=S_{i-1}(t)\oplus S_{i+1}(t)$

rule 102：$S_i(t+1)=S_i(t)\oplus S_{i+1}(t)$

rule 105：$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)\oplus S_{i+1}(t)\oplus 1$

rule 150：$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)\oplus S_{i+1}(t)$

rule 153：$S_i(t+1)=S_i(t)\oplus S_{i+1}(t)\oplus 1$

rule 165：$S_i(t+1)=S_{i-1}(t)\oplus +S_{i+1}(t)\oplus 1$

rule 170：$S_i(t+1)=S_{i+1}(t)$

rule 195：$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)\oplus 1$

rule 204：$S_i(t+1)=S_i(t)$

rule 240：$S_i(t+1)=S_{i-1}(t)$

rule 255：$S_i(t+1)=1$

其中，rule 60 就是国内熟悉的级间模 2 加。如果以 rule 60 为反馈关系构建单一 CA，则该 CA 就是级间模 2 加线性反馈移位寄存器。但线性 CA 的概念比级间模 2 加线性反馈移位寄存器广泛得多，可以用其他规则来构建单一 CA，也可以构建混合 CA，还可以在控制信号作用下，使反馈关系不断发生变化，来构建反馈关系动态变化的 PCA。

### 3．反馈关系动态变化的 PCA

在使用 LFSR 作为乱源时，就有在加乱过程中不断改变乱源反馈关系的加密思想。当 CA 作为乱源时，也有使 CA 反馈关系不断变化的加乱方式，这种 CA 就是前面提到的 PCA，它在实现上非常方便，特别适合用 VLSI 硬件实现。

有文献中提到可以用 rule 90 和 rule 150 来构建 PCA，因为

rule 90：$S_i(t+1)=S_{i-1}(t)\oplus S_{i+1}(t)$

rule 150：$S_i(t+1)=S_{i-1}(t)\oplus S_i(t)\oplus S_{i+1}(t)$

它们的差别仅在于每一级是否参与自身下一状态的生成，所以每级只需要由一个控制线来决定它是否作为自身反馈逻辑的输入，就可以使每级在不同时刻采用 rule 90 或 rule 150 两种不同反馈逻辑。对于 $n$ 级 CA，只要一个 $n$ 维向量，就可决定某一时刻该 CA 采用 $2^n$ 种状态转移变换中的哪一种，这种 PCA 的动态变化显然比 LFSR 更灵活。

### 4．CA 乱数发生器的各种非线性变化

序列密码中的各种非线性编码思想都可用于 CA 乱数发生器的构建：CA 可以不规则步进，也可以用非线性前馈逻辑（如查代替表）、乱数出口置换，还可以采用随时间变化的代替表等。采用 CA 作为乱源是 LFSR 的推广，所以采用 LFSR 作为乱源时的非线性化技术一般都可用于以 CA 作为乱源的加密。

在 ICICS’97 会议上，有人提出将 PCA 的某级输出放入缓冲器，经 $m$ 个时钟周期后，将 $m$ 比特经随时间变化的置换后输出作为乱数。

还有人提出一种在 GF($q$)上的 CA 乱数发生器，它由一个 $n$ 级 CA 和一个 $q$ 元 RAM 构成，用 $CA_i$ 表示 CA 的第 $i$ 级，RAM($x$)表示 RAM 中地址为 $x$ 的单元内容，RAM 中的内容是 0～$q$−1 的一个排列。每个时钟周期经下列步骤后输出 1 个 GF($q$)上的乱数符号。

（1）CA 进行一次状态转移。

（2）交换 RAM($CA_i$)和 RAM($CA_i$+$n$/2)，$i$=1,2,⋯,$n$/2（假定 $n$ 为偶数）。

（3）计算 $S=\sum_{i=1}^{r}\mathrm{RAM}(\mathrm{CA}_{it})$，其中∑表示模 $q$ 加，$i$ 和 $t$ 为常数，且 $rt\leqslant n$。

（4）如果 $S<(q-1)/2$，则 CA 再进行一次状态转移。

（5）计算 $k=\sum_{i=1}^{n}\mathrm{CA}_i$，其中∑表示模 $q$ 加。

（6）输出 RAM($k$)作为乱数。

此方案中 CA 采用不规则步进，非线性代替表 RAM 是随时间变化的，且输出经过了非线性查表。

### 5．对 GF(2)和 GF($q$)上一维线性 CA 的分析与综合

在这方面已发表了若干论文，这些论文从 CA 状态转移矩阵的特征多项式入手，对 CA 进行了分析，并讨论了由输出序列求初态的综合方法。另外，还给出了某些最长周期 CA 的状态转移矩阵。

加拿大的 K. Cattell 等人在 GF($q$)（$q$=2,3,5,7,11）上寻找了当式（3-12-2）中 $b_i$=1（1≤$i$≤$n$−1）、$c_j$=−1（2≤$j$≤$n$）、$d_k\in\{0,1\}$（1≤$k$≤$n$）、$c_1$=$b_n$=0（$n$≤40）时，$d_i$ 中“1”的个数最少，且能产生最长周期的线性 CA，它们的状态转移矩阵如表 3.12.1 所示。

表 3.12.1　不同 $n$ 和 $q$ 的状态转移矩阵

| $n$ \ $q$ | 2 | 3 | 5 | 7 | 11 |
|---|---|---|---|---|---|
| 3 | 1 | none | none | none | none |
| 4 | 1,3 | 1,4 | none | none | none |
| 5 | 1 | 1,3 | 1,3 | 1,2,3 | 1,3 |
| 6 | 1 | 1,6 | 1,2,5 | 1,2,5,6 | 1,2 |
| 7 | 3 | 1,2,3,5 | 1,3,4 | 1,3,5 | none |
| 8 | 2,3 | 1,4,5 | 1,3,6 | 1,2,5,6 | 1,6,7 |
| 9 | 1 | 1,3 | 1,2,5 | 1,2,7 | 1,3 |
| 10 | 2,7 | 1,6 | 1,2 | 1,2,3,1 | 3,4,5 |
| 11 | 1 | 1,9 | 1,9 | 1,2,7 | 1,9 |
| 12 | 3,7 | 1,8 | 3,4 | 1,3,8 | 3,6 |

续表

| n \ q | 2 | 3 | 5 | 7 | 11 |
|---|---|---|---|---|---|
| 13 | 5 | 1,7 | 3,5 | 3,7,9 | 1,11 |
| 14 | 1 | 1,6 | 3,4 | 1,4,12 | 1,8 |
| 15 | 3 | 1,13 | 1,7 | 1,4,7 | 1,7 |
| 16 | 1,15 | 1,2,3 | 3,6 | 1,4,14 | 1,2,9 |
| 17 | 5 | 3,5 | 1,2,17 | 1,2,15 | 5,9 |
| 18 | 1,17 | 1,2 | 1,6 | 1,3,18 | 3,6 |
| 19 | 3 | 1,5 | 1,3,4 | 1,3,17 | 7,9 |
| 20 | 2,3 | 1,2,15 | 1,4 | 1,2,18 | 1,2 |
| 21 | 1,10 | 3,5 | 1,9 | 1,2,15 | 1,5 |
| 22 | 5 | 1,18 | 1,2,7 | 1,6,22 | 3,8 |
| 23 | 1 | 3,19 | 5,15 | 1,9,13 | 5,13 |
| 24 | 8,12 | 1,8 | 1,18 | 1,3,24 | 1,6,11 |
| 25 | 9 | 1,15 | 3,15 | 1,2,9 | 3,7 |
| 26 | 1 | 1,6 | 1,16 | 1,3,10 | 1,2,5 |
| 27 | 1,20 | 3,7 | 1,3,22 | 1,2,7 | 1,23 |
| 28 | 3 | 3,14 | 7,8 | 1,4,16 | 1,4 |
| 29 | 1 | 1,11 | 1,15 | 1,2,15 | 1,11 |
| 30 | 1 | 1,4,25 | 1,12 | 1,4,16 | 1,2 |
| 31 | 11 | 1,5 | 5,15 | 1,7,13 | 1,9 |
| 32 | 1,15 | 5,12 | 3,8 | 1,2,32 | 1,4,17 |
| 33 | 1 | 3,9 | 3,13 | 1,4,5 | 5,13 |
| 34 | 1,19 | 1,6 | 1,2,17 | 1,4,8 | 1,26 |
| 35 | 1 | 1,13 | 1,31 | 1,2,15 | 7,9 |
| 36 | 6 | 3,30 | 3,32 | 1,2,6 | 5,20 |
| 37 | 9 | 1,23 | 3,9 | 1,3,9 | 7,17 |
| 38 | 7 | 1,2 | 1,10 | 1,3,12 | 7,10 |
| 39 | 1 | 1,9 | 1,7 | 1,3,9 | 1,11 |
| 40 | 8 | 3,30 | 17,18 | 1,6,12 | 1,30 |

例如，在表 3.12.1 中，当 $n$=6 和 $q$=5 时，GF(5)={1,2,5}，这表示 $d$=(1,1,0,0,1,0)，即状态转移矩阵为：

$$\boldsymbol{A}=\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

# 第4章

# P1363 公钥标准简介

IEEE 在 1997 年提出了一份公钥标准草案——P1363，现经多次修改，已公布到第 8 版，附有 6 个附录。公布该标准的目的是提供一个有关现有公钥密码技术的全面参考文件，供使用者选择。P1363 公钥标准的内容包括共同密钥的建立（Key Agreement）、消息的加密和脱密（Encryption/Decryption）、数字签名的生成和校验（Signature Generation/Verification），以及有关的技术，如 Hash 函数、掩码生成函数（Mask Generation Function，MGF）、填补（Padding）、密钥推导函数（Key Derivation Function）、消息编码/解码（message encoding/decoding）等。

P1363 公钥标准收纳的各种密码方案，按所依据的难题（Hard Problem）可划分为三类，离散对数（DL）、椭圆曲线上的离散对数（EC）和大数分解（IF）；按功能也可分为三类，消息的加/脱密方案、共同密钥生成方案和数字签名方案。

本节按各密码方案的功能介绍以下内容。

（1）一种消息加/脱密方案，即 RSA 方案（IF/ES），并给出了一种消息在加密前进行编码的方法，即 EME-OAEP-Hash 编码。

（2）六种共同密钥生成方案（见表 4.0.1），并要求在双方取得共同密钥后，再经 KDF X9.42 函数运算，得到实际使用的共同密钥。

表 4.0.1　六种共同密钥生成方案

| 数 学 基 础 | 离散对数 DL | 椭圆曲线上的离散对数 EC |
|---|---|---|
| 密钥生成方案 | DL/KAS/DH1<br>DL/KAS/DH2<br>DL/KAS/MQV | EC/KAS/DH1<br>EC/KAS/DH2<br>EC/KAS/MQV |

（3）十种数字签名方案和校验方法见表 4.0.2。

表 4.0.2　十种数字签名方案和校验方法

| 数 学 基 础 | 大数分解 IF | | 离散对数 DL | 椭圆曲线上的离散对数 EC |
|---|---|---|---|---|
| | 可　逆 | 不 可 逆 | | |
| 数字签名方案 | IF/SSR/RSA1<br>IF/SSR/RSA2<br>IF/SSR/RW | IF/SSA/RSA1<br>IF/SSA/RSA2<br>IF/SSA/RW | DJ/SSA/NR<br>DL/SSA/DSA | EC/SSA/NR<br>EC/SSA/DSA |
| 校验方法 | 签名前用<br>EMSR-ISO-9796 编码 | 签名前用<br>EMSA-X9.31-Hash 编码 | 签名前用<br>EMSA-Hash 编码 | |

## 4.1 RSA 方案（IF/ES）

消息的加密和脱密方案只有一种，即众多周知的 RSA 方案（IF/ES）。P1363 公钥标准规定，消息在加密之前需先经 EME-OAEP-Hash（Encoding Method for Encryption-Optional Asymmetric Encryption Padding-Hash）编码，方法如下：

选定某种 Hash 函数：P1363 公钥标准推荐 SHA-1（FIPS PUB 180-1）或 RIPEMD-160（ISO/IEC-10118-3 第 7 款）。

选定某种 MGF 函数：它与 Hash 函数的区别在于其输出长度每次可以自由给定，而 Hash 函数的输出长度是固定的。

MGF-Hash 函数是基于 Hash 函数的 MGF 函数，这里所用的 Hash 函数可以是 SHA、RIPEMD-160 或 P1363 公钥标准附录中的其他函数，将它记为 Hash，其输出长度为 hLen。MGF-Hash 函数的输入字节序列记为 $Z$，其长度记为 zLen，$zLen<2^{61}-4$。输出长度记为 oLen，$oLen \leqslant hLen \times 2^{32}$。

（1）P1363 公钥标准推荐的 MGF-Hash 函数可按如下步骤进行计算：

① 如果 $zLen \geqslant 2^{61}-4$ 或 $oLen > hLen \times 2^{32}$，则报“出错”并停止工作。

② 令 counter=0，cThreshold=oLen/hLen。

③ 令 $M$ 为空字节序列。

④ 将 counter 转换成长度为 4 字节的字节序列 $C$。

⑤ 用选定的 Hash 函数计算 Hash($Z \| C$)，生成长度为 hLen 的字节串 $H$。

⑥ 令 $M=M \| H$。

⑦ counter=counter+1，如果 counter<cThreshold，转第④步。

⑧ 输出 $M$ 的前 oLen 字节，作为 MGF 函数的输出。

（2）P1363 公钥标准还推荐了 OAEP，即可选择的非对称加密填补，其编码方法如下：

输入：

① 运行 Hash 函数 $F$，其输出长度记为 hLen（单位为字节，下同，特殊注明者除外）。

② 运行 MGF 函数 $G$；

③ 种子序列 seed，其长度记为 seedLen。

④ 编码参数 $P$（可以是空序列）。

⑤ 输出长度记为 oLen。

⑥ 产生消息 $M$，其长度记为 mLen，且 $mLen \leqslant oLen-hLen-seedLen-1$。

输出：经过编码的消息 EM，长度为 oLen。

运算：

① 若 mLen>oLen−hLen−seedLen−1，则报“出错”并停止工作。

② 令 $S$ 为全“0”字节序列，长度为 oLen−mLen−hLen−seedLen−1，$T$ 为 8 比特的字节 00000001。

③ 令 $M'=S \| T \| M$。

④ 对 $P$ 运行 $F$ 函数，产生 cHash，长度为 hLen。

⑤ 令 DB=cHash || $M'$，长度为 oLen−seedLen。

⑥ 对 seed 运行 $G$ 函数，产生 dbMask，长度为 oLen−seedLen。

⑦ 令 maskedDB=DB⊕dbMask。

⑧ 对 maskedDB 运行 $G$ 函数，产生 seedMask，长度为 seedLen。

⑨ 令 MaskedSeed=seed⊕seedMask。

⑩ 令 EM=maskedSeed || maskedDB。

⑪ 输出 EM。

（3）OAEP 的解码方法如下：

输入：

①～④同编码方法。

⑤ 经过编码的消息 EM，长度为 oLen。

输出：消息原文 $M$。

运算：

① 若 oLen<seedLen+hLen+1，报“出错”并停止工作。

② 令 maskedSeed 为 EM 中最左边的 seedLen 个字节，maskedDB 为其余字节。

③ 对 maskedDB 运行 $G$ 函数，产生字节序列 seedMask，长度为 seedLen。

④ 令 seed=maskedSeed⊕seedMask。

⑤ 对 seed 运行 $G$ 函数，产生字节序列 dbMask，长度为 oLen−seedLen。

⑥ 令 DB=maskedDB⊕dbMask。

⑦ 对参数 $P$ 运行 $F$ 函数，产生字节序列 cHash，长度为 hLen。

⑧ 将 DB 的前 hLen 个字节与 cHash 比较，若不相同，报“出错”，停止运行。

⑨ 令 $M'$为 DB 除前 hLen 个字节以外的其余部分。

⑩ 令 $T$ 为 $M'$最左边的一个非 0 字节，若 $T$≠00000001，报“出错”，停止运行。

⑪ 去掉 $T$ 及其以左的全部字节，余下的字节为 $M$。

⑫ 输出 $M$。

（4）EME-OAEP-Hash 的编码方法如下：

输入：

① 输出序列比特长度的上限 $l$；

② 消息 $M$，长度 mLen≤$l$/8–41 字节。

③ 编码参数 $P$（可以是空序列）。

输出：代表消息的整数 $f$≥0，长度≤$l$ 比特。

运算：

① 生成一个种子序列 seed，长度为 hLen。

② 以选定的 Hash 函数、MGF 函数、seed 序列、编码参数 $P$ 以及输出 oLen = $\lfloor l/8 \rfloor$ 对 $M$ 进行 OAEP 编码，产生 EM。

③ 将 EM 转换成整数 $f$。

④ 输出 $f$。

（5）EME-OAEP-Hash 的解码方法如下：

输入：

① 经过编码的消息长度的上限为 $\lfloor l/8 \rfloor$。

② 经过编码的消息 $f$≥0。

③ 编码参数 $P$。

输出：消息 $M$。

运算：

① 将 $f$ 转换成字节序列 EM，长度为 $\lfloor l/8 \rfloor$。

② 对 EM 进行 OAEP 解码运算，产生消息 $M$。

③ 输出 $M$。

## 4.2 共同密钥生成方案

共同密钥生成方案有 6 种，都要求在双方利用某种秘密数值推导算法（SVDP）取得共同的秘密数值后，再经过 KDF X9.42 函数运算，得出实际使用的共同密钥。P1363 公钥标准推荐的 SVDP 有以下 4 种，分别用于不同的共同密钥生成方案。

（1）DLSVDP-DH（Diffie-Hellman）。

在 GF($q$)中构建共同密钥生成方案。若在素域上构建，则 $q$ 为素数；若在有限域上构建，则 $q$=$2^m$。$g$ 为 GF($q$)中的一个元素，作为指数运算的底数；$r$ 为素数，$r|q-1$，$g$ 的阶为 $r$；$k$=($q$–1)/$r$；$s$ 为密钥，$w$=$g^s$ 为公钥。

输入：

① 自己的密钥 $s$。

② 对方的公钥 $w'$。

③ DL 的参数 $q$、$r$ 和 $g$，$(q-1)$除 $r$ 的余因子 $k$ 为可选参数。

④ 注明是否要进行余因子指数运算。

输出：导出的共同的秘密数值，它是一个非零域元素 $z \in \mathrm{GF}(q)$。

运算：

① 若要进行余因子指数运算，则令 $j=k$，否则令 $j=1$。

② 计算域元素 $z=\exp(w',js)$。注：$\exp(a,b)$表示 $a^b$，后同。

③ 输出 $z$。

附注：

① 进行余因子指数运算的目的是为了防御“小子群攻击”。对公钥 $w'$进行适当的检验也可达到相同的目的。

② 余因子 $k$ 只依赖 DL 参数，故可同参数一起存储，多次使用。

③ $z$ 是 GF($q$)的一个由 $g$ 的所有方幂（除 1 外）组成的子集中的一个元素，故不可能为 0。

（2）DLSVDP-MQV（Menez-Ou-Vanstone）。

$q$、$r$、$g$、$s$、$w$、$k$ 的定义同 DLSVDP-DH 方案，$u$ 为即时生成的密钥，$v=g^u$，是对应于 $u$ 的公钥。

输入：

① 自己的两对密钥$(s,w)$与$(u,v)$。

② 对方的两个公钥 $w'$与 $v'$。

③ DL 的参数 $q$、$r$ 和 $g$，以及 $l=\lceil (\log_2 r)/2 \rceil$，余因子 $k$ 为可选参数。

④ 注明是否要进行余因子指数运算。

输出：共同的秘密数值。

运算：

① 计算整数 $t=v \bmod 2^l$，令 $t=t+2^l$。

② 计算整数 $t'=v' \bmod 2^l$，令 $t'=t'+2^l$。

③ 若要进行余因子指数运算，则令 $j=k$，否则令 $j=1$。

④ 计算整数 $e=j(ts+u) \bmod r$。

⑤ 计算域元素 $z=\exp[v'\exp(w',t'),e]$。

⑥ 输出 $z$。

（3）ECSVDP-DH。

- $q$ 为椭圆曲线基域中元素的个数；
- $a,b \in \mathrm{GF}(q)$，是定义椭圆曲线 $E$ 的方程系数；
- $\#E$ 表示椭圆曲线 $E$ 中点的个数；
- $r$ 是素数，$r \mid \#E$；

- $G$ 是 $E$ 中的一个点，其阶为 $r$；
- $k=\#E/r$，是 $\#E$ 除 $r$ 的余因子；
- $s$ 为密钥；
- $w=sG$ 是公钥。

输入：

① 自己的密钥 $s$。

② 对方的公钥 $w'$。

③ EC 的参数 $q$、$a$、$b$、$r$ 和 $G$，余因子 $k$ 为可选参数。

④ 注明是否需要进行余因子乘法运算。

输出：$z$。

运算：

① 若要进行余因子乘法运算，则令 $j=k$，否则令 $j=1$。

② 计算椭圆曲线上一个点 $P=jsw'$。

③ 若 $P=0$，报“出错”，停止运算。

④ 若 $z=X_P$，即 $P$ 的 $X$ 坐标。

⑤ 输出 $z$。

（4）ECSVDP-MQV。

$u$ 为另一个密钥，$v=uG$ 是公钥，其他符号同前。

输入：

① 一方的两对密钥$(s,w)$及$(u,v)$，其中 $v=(x,y)$。

② 对方的两个公钥 $w'$及 $v'=(x',y')$。

③ EC 的参数 $q$、$a$、$b$、$r$ 和 $G$，余因子 $k$ 为可选参数。

④ 注明是否需要进行余因子乘法运算。

输出：$z$。

运算：

① 将 $x$ 转换成整数 $i$。

② 计算整数 $t=i \bmod 2^l$，令 $t=t+2^l$。

③ 将 $x'$转换成 $i'$。

④ 计算整数 $t'=i' \bmod 2^l$，令 $t'=t'+2^l$。

⑤ 若要进行余因子乘法，则令 $j=k$，否则令 $j=1$。

⑥ 计算整数 $e=j(ts+u) \bmod r$。

⑦ 计算椭圆曲线上一个点 $P=e(V'+t'w')$。

⑧ 若 $P=0$，报“出错”，停止运行。

⑨ 令 $z=X_P$，即 $P$ 的 $X$ 坐标。

⑩ 输出 z。

KDF X9.42 运算方法如下：

输入：

① Hash 函数，其输出长度为 hLen。

② 双方共同的秘密序列 $Z$，长度为 zLen。

③ 密码推导参数 $P$，长度为 pLen。注意：zLen+pLen$\leqslant 2^{61}$。

输出：共同密钥 $K$，长度为 hLen。

运算：

① 若 zLen+pLen$>2^{61}$，报“出错”并停止工作。

② 用函数 Hash 计算 $K$=Hash($Z \| P$)，长度为 hLen。

③ 输出 $K$。

利用上述 SVDP 及 KDF X9.42，可构成以下 6 种共同密钥生成方案。

（1）建立在 DL 上的 DH 方案（DL/KAS/DH1）。要求双方事先确定共同的 DL 参数、秘密数值推导算法 DLSVDP-DH 和密钥推导函数 KDF X9.42。

运算：

① 用 DLSVDP-DH 从自己的密钥 $s$ 和对方的公钥 $w'$推导出共同的秘密数值 $z$。

② 将 $z$ 转换成字节序列 $Z$。

③ 用密钥推导函数从 $Z$ 导出共同密钥。

（2）建立在 EC 上的 DH 方案（EC/KAS/DH1）。与 DL/KAS/DH1 的唯一差别，在于使用共同的 EC 参数和 ECSVDP-DH 来代替 DL 参数和 DLSVDP-DH。

（3）建立在 DL 上的 DH2 方案（DL/KAS/DH2）。要求双方事先确定两套 DL 参数，以及秘密数值推导算法 DLSVDP-DH 和密钥推导函数 KDF X9.42。

运算：

① 用 DLSV-DH 从自己的密钥 $s$ 和对方使用同一对 DL 参数的公钥 $w'$推导出共同的秘密数值 $z_1$，将 $z_1$ 转换成字节序列 $Z_1$。

② 用 DLSV-DH 从自己的另一个使用第二对 DL 参数的密钥 $u$ 和对方使用同一对 DL 参数的公钥 $v'$推到出共同的秘密数值 $z_2$，将 $z_2$ 转换成字节序列 $Z_2$。

③ 令 $Z=Z_1 \| Z_2$。

④ 用密钥推导函数从 $Z$ 导出共同密钥。

（4）建立在 EC 上的 DH2 方案（EC/KAS/DH2）。同 DL/KAS/DH2 的唯一差别在于使用两套 EC 参数及 ECSVDP-DH 来代替两套 DL 参数及 DLSVDP-DH。

（5）建立在 DL 上的 MQV 方案（DL/KAS/MQV）。要求双方事先确定共同的 DL 参数、

DLSVDP-MQV 以及密钥推导函数 KDF X9.42。

运算：

① 用 DLSVDP-MQV 从一方自己的两对密钥$(s,w)$与$(u,v)$和对方的两个公钥$w'$与$v'$推导出共同的秘密数值 $z$。

② 将 $z$ 转换成字节序列 $Z$。

③ 用密钥推导函数从 $Z$ 导出共同密钥。

附注：如果两对密钥中一对是经过验证的，一对是一次性的，可以得到优化的 DL/KAS/MQV。

（6）建立在 EC 上的 MQV 方案（EC/KAS/MQV）。同 DL/KAS/MQV 的唯一差别在于使用 EC 参数和 ECSVDP-MQV 来代替 DL 参数和 DLSVDP-MQV。

## 4.3 数字签名方案和校验方法

数字签名方案和校验方法有 10 种方案，可分为两类：第一类有 6 种，建立在 IF 之上；第二类有 4 种，建立在 DL 或 EC 之上。

### 4.3.1 建立在 IF 之上的可逆方案

第一类前 3 种方案都是可逆的，即校验方法可以从数字签名方案逆推出来。

在进行数字签名（简称签名）之前，要求消息先经 EMSR-ISO-9796（Encoding Method for Signature with Message Recovery，用于可逆签名的编码方法）进行编码。

EMSR-ISO-9796 编码方法如下：

输入：

① 规定输出比特长度的上限 $l$。

② 消息 $M$，长度 mLen≤$(l+3)/16$。

输出：经过编码的消息 $f$。

运算：

① 若 $M$ 的长度大于$(l+3)/16$，则报“出错”，停止运行。

② 将 $M$ 转换成比特序列 MB，输入格式化程序（见 ISO/IEC 9796 第一版 5.1～5.4），产生比特序列 IR，长度为 $l$ 比特。

③ 将 IR 转换成整数 $f$。

④ 输出 $f$。

EMSR-ISO-9796 解码方法如下：

输入：

① 经过编码的消息的比特长度上限。

② 经过编码的消息 $f \geqslant 0$。

输出：消息 $M$。

运算：

① 判断 $f \equiv 6 \bmod 16$，是则继续执行②，否则报“出错”，停止运行。

② 判断 $f$ 的比特长度=$l$（即 $f \geqslant 2^{l-1}$、$f<2^{l}$），是则继续执行③，否则报“出错”，停止运行。

③ 将 $f$ 转换成比特序列 IR′，长度为 $l$，作为“中间整数”，输入校验程序（见 ISO-/IEC 9796 第一版 6.2～6.2）后产生消息 $M$。

④ 输出 $M$。

附注：这一方法的好处是，它保证输出$\equiv 6 \bmod 16$，从而适用于 P1363 公钥标准所推荐的三种可逆签名方案中的任意一种。

### 1．RSA 方案（IF/SSR/RSA1）

（1）运算方法如下：

输入：

① 需要签名的消息 $M$。

② 签名者的密钥（$n$,$d$）。

③ 编码程序 EMSR-ISO-9796。

输出：对 $M$ 的签名。

运算：

① 对 $M$ 进行编码，产生 $f<n$。

② 计算 $s=\exp(f,d) \bmod n$。

③ 输出 $s$。

（2）校验方法如下：

① 签名 $s$。

② 签名者的公钥（$n$,$e$）。

③ 编码程序 EMSR-ISO-9796。

输出：消息 $M$。

运算：

① 计算 $f=\exp(s,e) \bmod n$。

② 对 $f$ 进行解码，产生 $M$。

③ 输出 $M$。

### 2．IF/SSR/RSA2 方案

该方案与 RSA 方案略有不同，要求编码后的消息$\equiv 6 \bmod 16$。

（1）运算方法如下：

输入与输出同 RSA 方案。

运算：

① 对 $M$ 进行编码，产生 $f<n$。

② 计算 $t=\exp(f,d) \bmod n$。

③ 令 $s\equiv\min(t,n-t)$。

④ 输出 $s$。

校验方法如下：

输入与输出同 RSA 方案。

运算：

① 计算 $t=\exp(s,e) \bmod n$。

② 若 $t\equiv 6 \bmod 16$，令 $f=t$，否则令 $f=n-t$。

③ 对 $f$ 进行解码，产生 $M$。

④ 输出 $M$。

### 3．RW 方案（IF/SSR/RW）

要求编码后的消息$\equiv 6 \bmod 16$。

（1）运算方法如下：

输入与输出同 RSA 方案。

运算：

① 对 $M$ 进行编码，产生 $f<n$。

② 对 Jacobi $[f/n]=+1$，令 $u=f$，否则 $u=f/2$。

③ 计算 $t=(f,d) \bmod n$。

④ 令 $s=\min(t,n-t)$。

⑤ 输出 $s$。

（2）校验方法如下：

输入与输出同 RSA 方案。

运算：

① 计算 $t_1=\exp(s,e) \bmod n$。

② 计算 $t_2=n-t_1$。

③ 若 $t_1\equiv 6 \bmod 16$，则令 $f=t_1$。

④ 否则若 $t_1\equiv 3 \bmod 8$，则令 $f=2t_1$。

⑤ 否则若 $t_2\equiv 6 \bmod 16$，则令 $f=t_2$。

⑥ 否则若 $t_2$≡3 mod 8，则令 $f$=2$t_2$。

⑦ 否则报“无效”，停止运行。

⑧ 对 $f$ 进行解码，产生 $M$。

⑨ 输出 $M$。

### 4.3.2　建立在 IF 之上的不可逆方案

第一类后 3 种是不可逆的，即校验签名时不用解码的办法，而是将原消息经过编码后再与校验结果相比较，所用编码方法是 EMSA-X9.31-Hash（Encoding Method for Signature with Appendix，用于不可逆签名的编码方案）。

（1）运算方法如下：

输入：

① 消息 $M$，字节度小于或等于 $2^{61}-1$。

② 输出的比特长度上限 $l$ 大于或等于 191。

③ 一种 Hash 函数 $F$（SHA-1 或 RIPEMD-160）。

输出：经过编码的消息 $f$。

运算：

① 用 $F$ 计算 $H$=Hash($M$)，字节长度为 20。

② 若 $M$ 的长度=0，令 $P_1$=4b（十六进制），否则令 $P_1$=6b（十六进制）。

③ 令 $P_2$ 是 bb（十六进制）构成的字节序列，长度为 $\lfloor (l+1)/8 \rfloor - 24$ 个字节。

④ 若 $F$ 为 SHA-1，令 $P_3$=33（十六进制）。若 $F$ 为 RIPEMD-160，令 $P_3$=31（十六进制）。

⑤ 令 $T$=$P_1 \| P_2 \| \text{ba} \| H \| P_3 \| \text{cc}$，其中 ba 与 cc 均为十六进制的字节。

⑥ 将 $T$ 转换成整数 $f$。

（2）验方法如下：

输入：

① 消息 $M$，字节长度小于 $2^{61}$。

② 经过编码的消息比特长度的上限 $l$ 大于或等于 191。

③ 经过编码的消息 $f$（整数，≥0）。

输出：“有效”或“无效”。

运算：

① 将 $f$ 转换成字节序列 $T$，长度为 $\lfloor (l+1)/8 \rfloor$。

② 若 $M$ 的长度为 0，令 $P_1$=4b（十六进制），否则令 $P_1$=6b（十六进制）。

③ 判断 $T$ 最左边的字节是否为 $P_1$，后面 $\lfloor (l+1)/8 \rfloor$−24 个字节是否均为 bb，再判断下一个字节是否为 ba（十六进制），否则报“无效”，停止运行。

④ 若 $F$ 为 SHA-1，判断最右边第二个字节是否为 33。若 $F$ 为 RIPEMD-160，则判断最右边的字节是否为 31（均为十六进制）。如果不是，则报“无效”，停止运行。

⑤ 判断最右边的一个字节是否为 cc，如果不是，则报“无效”，停止运行。

⑥ 去掉 $T$ 的最左边的 $\lfloor (l+1)/8 \rfloor - 22$ 个字节和最右边的 2 个字节，余下字节序列 $H'$，长度为 20。

⑦ 计算 $H=F(M)$，长度为 20。

⑧ 若 $H=H'$，则输出“有效”，否则输出“无效”。

### 1．IF/SSA/RSA1

（1）运算方法同 RSA 方案，只是把输入中的③改为“编码程序 EMSA-X9.31-Hash”。

（2）校验程序如下：

输入：

① 签名 $s$。

② 签名者的公钥$(n,e)$。

③ 编码程序 EMSA-X9.31-Hash。

④ 消息 $M$。

输出：“有效”或“无效”。

运算：

① 计算 $f=\exp(s,e) \bmod n$。

② 判断 $f$ 是否为经过编码的消息 $M$，若是，则输出“有效”。

③ 否则输出“无效”。

### 2．IF/SSA/RSA2

同 IF/SSR/RSA2，但要求编码后的消息≡12 mod 16。

（1）运算方法同 IF/SSR/RSA2，只是输入中的③改为“编码程序 EMSA-X9.31-Hash”。

（2）校验方法如下：

输入和输出同 IF/SSR/RSA1 的校验方法。

运算：

① 计算 $t=\exp(s,e) \bmod n$。

② 若 $t\equiv 12 \bmod 16$，令 $f=t$，否则令 $t=n-t$。

③ 判断 $f$ 是否为经过编码的消息 $M$，若是，则输出“有效”。

④ 否则输出“无效”。

### 3．IF/SSA/RW

同 IF/SSR/RW 方案，但要求编码后的消息≡12 mod 16。

（1）运算方法同 IF/SSR/RW 方案，只是输入③改为“编码程序 EMSA-X9.31-Hash”。

（2）校验方法如下：

输入和输出同 IF/SSR/RSA1 的校验方法。

运算：

①～⑦同 IF/SSR/RW。

⑧ 判断 $f$ 是否为经过编码的消息 $M$，若是，则输出“有效”。

⑨ 否则输出“无效”。

### 4.3.3　建立在 DL 或 EC 之上的不可逆方案

第二类 4 种都是不可逆的，它们都要求消息先经过 EMSA-Hash 编码。

EMSA-Hash 也是用于不可逆签名的编码方法。

（1）运算方法如下：

输入：

① 消息 $M$，字节长度小于或等于 $2^{61}-1$。

② 输出的比特长度上限 $l$。

③ Hash 函数 SHA 或 RIPEMD。

输出：经过编码的消息 $f$，其比特长度≤min(160,$l$)。

运算：

① 计算 $H$=Hash($M$)，比特长度为 160。

② 若 $l$<160，去掉 $H$ 最右边的 160–$l$ 比特，再将余下的 $l$ 比特转换成整数 $f$。

③ 否则将 $H$ 转换成整数 $f$。

④ 输出 $f$。

（2）校验方法是用同样的编码方法对消息 $M$ 进行计算，可得整数 $g$，检验是否 $g$=$f$。

#### 1．DL/SSA/NR（Nyberg-Rueppel）

该方案是建立在 DL 上的 NR 方案，设 $g$ 是有限域 GH($q$)中的一个元素，$r$ 是素数，$r|q-1$，$g$ 的阶为 $r$，$s$ 为密钥，$w$ 为公钥，$w=g^s$。

（1）运算方法如下：

输入：

① 签名者的密钥 $s$。

② 双方共用 DL 的参数 $q$、$r$ 和 $g$。

③ 编码方法 EMSA-Hash。

④ 消息 $M$。

输出：对 $M$ 的签名（$c$,$d$）。

运算：

① 对消息 $M$ 进行编码，产生整数 $f$，$0 \leqslant f < r$。

② 由 $q$、$r$ 和 $g$ 产生一对一次性的密钥（$u$,$v$），即随机生成 $u$，在 $GF(q)$中计算 $v=g^u$。

③ 将 $v$ 转换成整数 $i$。

④ 计算整数 $c=i+f \bmod r$，若 $c=0$，则回到①。

⑤ 计算整数 $d=u-sc \bmod r$。

⑥ 输出($c$,$d$)。

（2）校验方法如下：

输入：

① 签名者的公钥 $w$。

② 双方共用 DL 的参数 $q$、$r$ 和 $g$。

③ 编码方法 EMSA-Hash。

④ 消息 $M$。

⑤ 对 $M$ 的签名。

输出：“有效”或“无效”。

运算：

① 判断 $c$ 是否属于[1,$r$–1]，以及 $d$ 是否属于[0,$r$–1]，如果不属于，则报“无效”，停止运行。

② 计算域元素 $j=\exp(g,d)\times\exp(w,c)$。

③ 将 $j$ 转换成整数 $i$。

④ 计算整数 $f=c-i \bmod r$。

⑤ 判断 $f$ 是否为经过 EMSA-Hash 编码的消息 $M$，若是，则输出“有效”。

⑥ 否则输出“无效”。

### 2．EC/SSA/NR

该方案是建立在 EC 上的 NR 方案。

（1）运算方法如下：

输入与输出同 DL/SSA/NR 方案，只是用 EC 的参数 $q$、$a$、$b$、$r$ 和 $G$ 代替 DL 的参数，参数的定义同 4.2 节。

运算：

① 对消息 $M$ 进行编码，产生整数 $f$，$0 \leqslant f < r$。

② 由 $q$、$a$、$b$、$r$ 和 $G$ 产生一对一次性的密钥($u$,$v$)，$u$ 为密钥，$v=uG$ 为公钥，$v=(x_v,y_v)$。

③ 将 $x_v$ 转换成整数 $i$。

④ 计算整数 $c=i+f \bmod r$，若 $c=0$，则回到①。

⑤ 计算整数 $d=u-sc \bmod r$。

⑥ 输出$(c,d)$。

（2）校验方法如下：

输入与输出同 DL/SSA/NR 方案的校验方法，只是用 EC 的参数 $q$、$a$、$b$、$r$ 和 $G$ 代替 DL 的参数。

运算：

① 判断 $c$ 是否属于$[1,r-1]$，以及 $d$ 是否属于$[0,r-1]$，如果不属于，则报“无效”，停止运行。

② 计算椭圆曲线上一个点 $P=dG+cW$，若 $P=0$，则输出“无效”，停止运行；否则令 $P=(x_P,y_P)$。

③ 将域元素 $x_P$ 转换成整数 $i$。

④ 计算整数 $f=c-i \bmod r$。

⑤ 判断 $f$ 是否为经过 EMSA-Hash 编码的消息 $M$，若是，则输出“有效”；否则输出“无效”。

### 3．DL/SSA/DSA

该方案是建立在 DL 上的 DSA 方案。

（1）运算方法如下：

输入与输出同 DL/SSA/NR 方案。

运算：

① 对消息 $M$ 进行编码，产生整数 $f$。

② 由 $q$、$r$ 和 $g$ 产生一对一次性的密钥$(u,v)$。

③ 将 $v$ 转换成整数 $i$。

④ 计算整数 $c=i \bmod r$，若 $c=0$，则回到①。

⑤ 计算整数 $d=u^{-1}(f+sc) \bmod r$，若 $d=0$，则回到①。

⑥ 输出$(c,d)$。

（2）校验方法如下：

输入与输出同 DL/SSA/NR 方案。

运算：

① 判断 $c$ 是否属于$[1,r-1]$，以及 $d$ 是否属于$[1,r-1]$，如果不属于，则报“无效”，停止运行。

② 对消息 $M$ 进行编码，产生 $f$。

③ 计算整数 $h=d^{-1} \bmod r$，$h_1=fh \bmod r$，$h_2=ch \bmod r$。

④ 计算域元素 $j=\exp(g,h_1)\times\exp(w,h_2)$。

⑤ 将域元素 $j$ 转换成整数 $i$。

⑥ 计算整数 $c'=i \bmod r$。

⑦ 若 $c'=c$，则输出“有效”；否则输出“无效”。

### 4．EC/SSA/DSA

该方案是建立在 EC 上的 DSA 方案。

（1）运算方法如下：

输入与输出同 EC/SSA/NR 方案。

运算：

① 对消息 $M$ 进行编码，产生 $f$。

② 由 $q$、$a$、$b$、$r$ 和 $G$ 产生一对一次性的密钥（$u$,$v$），$u$ 为密钥，$v=uG$ 为公钥，其中 $v=(x_v,y_v)$。

③ 将 $x_v$ 转换成整数 $i$。

④ 计算整数 $c\equiv i \bmod r$，若 $c=0$，则回到①。

⑤ 计算整数 $d=u^{-1}(f+sc) \bmod r$，若 $d=0$，则回到①。

⑥ 输出($c$,$d$)。

（2）校验方法如下：

输入与输出同 EC/SSA/NR 方案的校验方法。

运算：

① 判断 $c$ 是否属于[1,$r$–1]，以及 $d$ 是否属于[1,$r$–1]，如果不属于则报“无效”，停止运行。

② 对消息 $M$ 进行编码，产生 $f$。

③ 计算整数 $h=d^{-1} \bmod r$，$h_1=fh \bmod r$，$h_2=ch \bmod r$。

④ 计算椭圆曲线上一个点 $P=h_1G+h_2W$，若 $P=0$，则输出“无效”，停止运行；否则令 $P=(x_P,y_P)$。

⑤ 将域元素 $x_P$ 转换成整数 $i$。

⑥ 计算整数 $c'=i \bmod r$。

⑦ 若 $c'=c$，则输出“有效”；否则输出“无效”。

## 4.4 密钥交换算法 KEA

KEA 是一种采用公钥技术的密钥交换算法，它以 Diffie-Hellman 的指数密钥交换协议为基础，并采用 SKIPJACK 加密算法将公钥交换生成的密钥压缩到 80 比特，作为消息加/脱密的密钥。

KEA 算法要求每个用户能够对获得的公钥进行真实性的验证，但对具体方法未做出统一

规定。KEA 具有与 SKIPJACK 同等的安全性，采用 160 比特长的指数作为用户的密钥，用于进行模指数运算。

KEA 算法需要下列数值：

- $p$：1024 比特素数，作为模数。
- $q$：160 比特素数，作为 $p-1$ 的因子。
- $g$：1024 比特，作为指数运算的底数，其模 $p$ 的阶是 $q$。
- $x$：160 比特，作为用户的秘密参数（$0<x<q$）。
- $Y$：1024 比特，对应于 $x$ 的公开参数，$Y=g^x \bmod p$。
- $r$：160 比特，作为随机数。
- $c$：80 比特常数，用十六进制表示为 72f1a87e92824198ab0b。

下面给出用户 A 与 B 双方进行密钥交换的过程（用下角标 A 和 B 来区分两个不同用户的数值）。

（1）A、B 双方交换公钥或从用户一览表中获取对方终端的证书，从证书中得到对方的公钥 $Y$、用户身份标识和其他有关信息。

（2）每一方验证得到的对方公钥 $Y$ 的真实性，确认公钥是真实用户的公钥。如果验证失败，过程即终止；反之进入（4）。

（3）双方交换公开的随机数。A 方生成 160 比特秘密随机数 $r_A$，计算 $R_A = g^{r_A} \bmod p$。B 方生成 160 比特秘密随机数 $r_B$，计算 $R_B = g^{r_B} \bmod p$。$R_A$ 和 $R_B$ 是 1024 比特的随机数，双方交换公开的数值 $R_A$ 和 $R_B$。

（4）双方在收到对方的公钥和公开的随机参数后，分别检查这些参数的阶是否为 $q$。双方需进行如下的计算和校验：

A 方：

$$1<R_B \text{ 和 } Y_B<p，(R_B)^q=1 \bmod p$$
$$(Y_B)^q=1 \bmod p$$

B 方

$$1<R_A \text{ 和 } Y_A<p，(R_A)^q=1 \bmod p$$
$$(Y_A)^q=1 \bmod p$$

若校验成立，进入（5），否则终止过程。

（5）A 方用 $Y_B$ 计算 $t_{AB}$，B 方用收到随机数 $R_A$ 计算 $t_{BA}$。

$$t_{AB} = (Y_B)^{r_A} \bmod p = g^{x_B r_A} \bmod p$$
$$t_{BA} = (R_A)^{x_B} \bmod p = g^{r_A x_B} \bmod p = g^{x_B r_A} \bmod p$$

（6）双方用（5）中类似的方法计算 $u$。

$$u_{AB}=(Y_A)^{r_B} \bmod p=g^{x_A r_B} \bmod p$$

$$u_{BA}=(R_B)^{x_A} \bmod p=g^{r_B x_A} \bmod p=g^{x_A r_B} \bmod p$$

（7）双方计算 $w$ 值，并检查 $w$ 是否不等于 0，即

$$w=(t+u) \bmod p \neq 0$$

如果检查通过，则进入（8），否则终止计算。

（8）将 $w$ 高位的 160 比特分成两部分，分别记为 $v_1$ 和 $v_2$，$v_1$ 为 $w$ 的最高位 80 比特，$v_2$ 为 $w$ 的次高位 80 比特。若将 1024 比特的 $w$ 从高位至低位按 1023～0 进行编号，则 $v_1$ 为第 1023～944 比特，$v_2$ 为第 943～864 比特。

（9）由 $v_1$ 和 $v_2$ 按照图 4.4.1 所示的逻辑生成 80 比特的消息加密密钥。

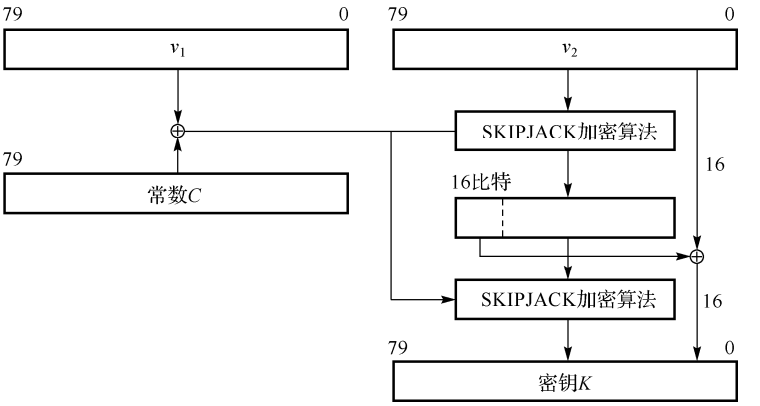



图 4.4.1　生成消息加密密钥的逻辑框图

用 $E[\ ]$表示以 $v_1 \oplus c$ 为密钥，用 SKIPJACK 加密算法进行加密，最终得到的消息加密密钥为：

$$K=(E[E[_{64}(v_2)]]) \parallel \{_{16}\{E[_{64}(V_2)]\} \oplus (v_2)_{16}\}$$

式中，$_{64}(\ )$表示高位 64 比特，$_{16}(\ )$表示高位 16 比特，$(\ )_{16}$ 表示低位 16 比特。

A 方和 B 方之间的 KEA 交换关系如图 4.4.2 所示。

KEA 在电子邮件加密中的应用。若在电子邮件（E-mail）加密中应用 KEA 算法，则收方不参与密钥的形成。这时需将 KEA 算法步骤（6）中的随机数 $R_B$ 用收方的公钥 $Y_B$ 代替，其他均不变。

电子邮件发方（A 方）、收方（B 方）按以下步骤计算密钥。

（1）A 方计算密钥过程。

① 从用户一览表中得到 B 方的证书，由证书可得到 B 方的公钥 $Y_B$、用户身份标识和其他信息。

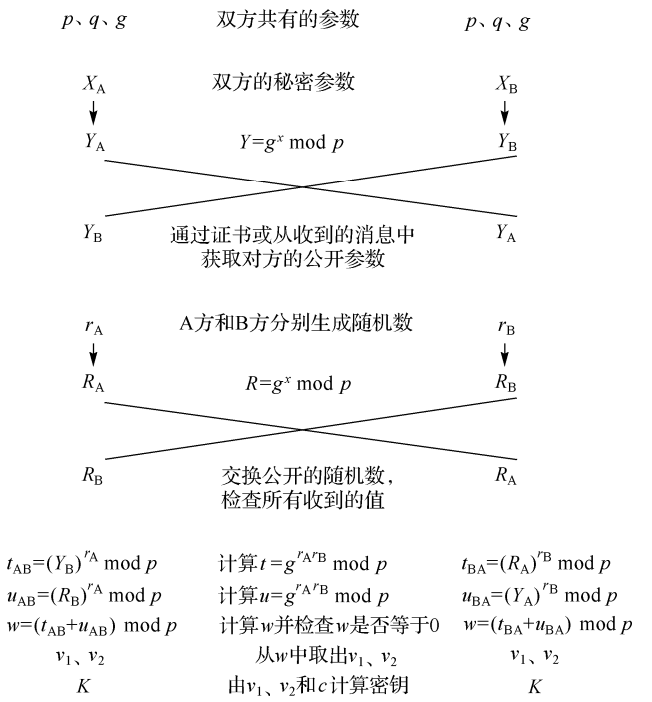



图 4.4.2　A 方和 B 方之间 KEA 交换关系图

② 检验 $Y_B$ 的真实性。

③ 校验 $1<Y_B<p$，且$(Y_B)^q=1 \bmod p$。

④ 生成 $r_A$，计算 $R_A=g^{r_A} \bmod p$，将 $R_A$ 放在消息中发给 B 方。

⑤ 计算 $t_{AB}=(Y_B)^{r_A} \bmod p=g^{r_Ax_B} \bmod p$。

⑥ 用 $Y_B$ 代替 KEA 算法步骤（6）中的 $R_B$，计算

$$u_{AB}=(Y_B)^{x_A} \bmod p=g^{x_Bx_A} \bmod p=g^{x_Ax_B} \bmod p$$

⑦ 计算 $w=(t_{AB}+u_{AB}) \bmod p \neq 0$。

⑧ 从 $w$ 中选取 $v_1$ 和 $v_2$。

⑨ 由 $v_1$ 和 $v_2$ 计算得到 80 比特的密钥 $K$。

（2）B 方计算密钥过程。

① B 方在收到电子邮件后，从用户一览表中得到 A 方的证书，由证书可得到 A 方的公钥 $Y_A$、用户身份标识和其他信息。

② 检验 $Y_A$ 的真实性。

③ 从电子邮件中得到 A 方的随机数 $R_A$。

④ 计算和校验 $1<R_A$ 和 $Y_A<p$，且$(R_A)^q=1 \bmod p$，$(Y_A)^q=1 \bmod p$。

⑤ 计算 $t_{BA}=(R_A)^{r_B} \bmod p=g^{r_A x_B} \bmod p$ 。

⑥ 用 $x_B$ 代替 KEA 算法步骤（6）中的 $r_B$，计算

$$u_{BA}=(Y_A)^{x_B} \bmod p=g^{x_A x_B} \bmod p$$

⑦ 计算 $w=(t_{AB}+u_{AB}) \bmod p \neq 0$。

⑧ 从 $w$ 中选取 $v_1$ 和 $v_2$。

⑨ 由 $v_1$ 和 $v_2$ 计算得到 80 比特的密钥 $K$。

# 附录 A

# *S* 盒

```
WORD Sbox[ ] = {
0x09d0c479, 0x28c8ffe0, 0x84aa6c39, 0x9dad7287, 0x7dff9be3, 0xd4268361,
0xc96da1d4, 0x7974cc93, 0x85d0582e, 0x2a4b5705, 0x1ca16a62, 0xc3bd279d,
0x0f1f25e5, 0x5160372f, 0xc695c1fb, 0x4d7ff1e4, 0xae5f6bf4, 0x0d72ee46,
0xff23de8a, 0xb1cf8e83, 0xf14902e2, 0x3e981e42, 0x8bf53eb6, 0x7f4bf8ac,
0x83631f83, 0x25970205, 0x76afe784, 0x3a7931d4, 0x4f846450, 0x5c64c3f6,
0x210a5f18, 0xc6986a26, 0x28f4e826, 0x3a60a81c, 0xd340a664, 0x7ea820c4,
0x526687c5, 0x7eddd12b, 0x32a11d1d, 0x9c9ef086, 0x80f6e831, 0xab6f04ad,
0x56fb9b53, 0x8b2e095c, 0xb68556ae, 0xd2250b0d, 0x294a7721, 0xe21fb253,
0xae136749, 0xe82aae86, 0x93365104, 0x99404a66, 0x78a784dc, 0xb69ba84b,
0x04046793, 0x23db5c1e, 0x46cae1d6, 0x2fe28134, 0x5a223942, 0x1863cd5b,
0xc190c6e3, 0x07dfb846, 0x6eb88816, 0x2d0dcc4a, 0xa4ccae59, 0x3798670d,
0xcbfa9493, 0x4f481d45, 0xeafc8ca8, 0xdb1129d6, 0xb0449e20, 0x0f5407fb,
0x6167d9a8, 0xd1f45763, 0x4daa96c3, 0x3bec5958, 0xababa014, 0xb6ccd201,
0x38d6279f, 0x02682215, 0x8f376cd5, 0x092c237e, 0xbfc56593, 0x32889d2c,
0x854b3e95, 0x05bb9b43, 0x7dcd5dcd, 0xa02e926c, 0xfae527e5, 0x36a1c330,
0x3412e1ae, 0xf257f462, 0x3c4f1d71, 0x30a2e809, 0x68e5f551, 0x9c61ba44,
0x5ded0ab8, 0x75ce09c8, 0x9654f93e, 0x698c0cca, 0x243cb3e4, 0x2b062b97,
0x0f3b8d9e, 0x00e050df, 0xfc5d6166, 0xe35f9288, 0xc079550d, 0x0591aee8,
0x8e531e74, 0x75fe3578, 0x2f6d829a, 0xf60b21ae, 0x95e8eb8d, 0x6699486b,
0x901d7d9b, 0xfd6d6e31, 0x1090acef, 0xe0670dd8, 0xdab2e692, 0xcd6d4365,
0xe5393514, 0x3af345f0, 0x6241fc4d, 0x460da3a3, 0x7bcf3729, 0x8bf1d1e0,
0x14aac070, 0x1587ed55, 0x3afd7d3e, 0xd2f29e01, 0x29a9d1f6, 0xefb10c53,
0xcf3b870f, 0xb414935c, 0x664465ed, 0x024acac7, 0x59a744c1, 0x1d2936a7,
0xdc580aa6, 0xcf574ca8, 0x040a7a10, 0x6cd81807, 0x8a98be4c, 0xaccea063,
```

```
0xc33e92b5, 0xd1e0e03d, 0xb322517e, 0x2092bd13, 0x386b2c4a, 0x52e8dd58,
0x58656dfb, 0x50820371, 0x41811896, 0xe337ef7e, 0xd39fb119, 0xc97f0df6,
0x68fea01b, 0xa150a6e5, 0x55258962, 0xeb6ff41b, 0xd7c9cd7a, 0xa619cd9e,
0xbcf09576, 0x2672c073, 0xf003fb3c, 0x4ab7a50b, 0x1484126a, 0x487ba9b1,
0xa64fc9c6, 0xf6957d49, 0x38b06a75, 0xdd805fcd, 0x63d094cf, 0xf51c999e,
0x1aa4d343, 0xb8495294, 0xce9f8e99, 0xbffcd770, 0xc7c275cc, 0x378453a7,
0x7b21be33, 0x397f41bd, 0x4e94d131, 0x92cc1f98, 0x5915ea51, 0x99f861b7,
0xc9980a88, 0x1d74fd5f, 0xb0a495f8, 0x614deed0, 0xb5778eea, 0x5941792d,
0xfa90c1f8, 0x33f824b4, 0xc4965372, 0x3ff6d550, 0x4ca5fec0, 0x8630e964,
0x5b3fbbd6, 0x7da26a48, 0xb203231a, 0x04297514, 0x2d639306, 0x2eb13149,
0x16a45272, 0x532459a0, 0x8e5f4872, 0xf966c7d9, 0x07128dc0, 0x0d44db62,
0xafc8d52d, 0x06316131, 0xd838e7ce, 0x1bc41d00, 0x3a2e8c0f, 0xea83837e,
0xb984737d, 0x13ba4891, 0xc4f8b949, 0xa6d6acb3, 0xa215cdce, 0x8359838b,
0x6bd1aa31, 0xf579dd52, 0x21b93f93, 0xf5176781, 0x187dfdde, 0xe94aeb76,
0x2b38fd54, 0x431de1da, 0xab394825, 0x9ad3048f, 0xdfea32aa, 0x659473e3,
0x623f7863, 0xf3346c59, 0xab3ab685, 0x3346a90b, 0x6b56443e, 0xc6de01f8,
0x8d421fc0, 0x9b0ed10c, 0x88f1a1e9, 0x54c1f029, 0x7dead57b, 0x8d7ba426,
0x4cf5178a, 0x551a7cca, 0x1a9a5f08, 0xfcd651b9, 0x25605182, 0xe11fc6c3,
0xb6fd9676, 0x337b3027, 0xb7c8eb14, 0x9e5fd030,
0x6b57e354, 0xad913cf7, 0x7e16688d, 0x58872a69, 0x2c2fc7df, 0xe389ccc6,
0x30738df1, 0x0824a734, 0xe1797a8b, 0xa4a8d57b, 0x5b5d193b, 0xc8a8309b,
0x73f9a978, 0x73398d32, 0x0f59573e, 0xe9df2b03, 0xe8a5b6c8, 0x848d0704,
0x98df93c2, 0x720a1dc3, 0x684f259a, 0x943ba848, 0xa6370152, 0x863b5ea3,
0xd17b978b, 0x6d9b58ef, 0x0a700dd4, 0xa73d36bf, 0x8e6a0829, 0x8695bc14,
0xe35b3447, 0x933ac568, 0x8894b022, 0x2f511c27, 0xddfbcc3c, 0x006662b6,
0x117c83fe, 0x4e12b414, 0xc2bca766, 0x3a2fec10, 0xf4562420, 0x55792e2a,
0x46f5d857, 0xceda25ce, 0xc3601d3b, 0x6c00ab46, 0xefac9c28, 0xb3c35047,
0x611dfee3, 0x257c3207, 0xfdd58482, 0x3b14d84f, 0x23becb64, 0xa075f3a3,
0x088f8ead, 0x07adf158, 0x7796943c, 0xfacabf3d, 0xc09730cd, 0xf7679969,
0xda44e9ed, 0x2c854c12, 0x35935fa3, 0x2f057d9f, 0x690624f8, 0x1cb0bafd,
0x7b0dbdc6, 0x810f23bb, 0xfa929a1a, 0x6d969a17, 0x6742979b, 0x74ac7d05,
0x010e65c4, 0x86a3d963, 0xf907b5a0, 0xd0042bd3, 0x158d7d03, 0x287a8255,
0xbba8366f, 0x096edc33, 0x21916a7b, 0x77b56b86, 0x951622f9, 0xa6c5e650,
```

```
0x8cea17d1, 0xcd8c62bc, 0xa3d63433, 0x358a68fd, 0x0f9b9d3c, 0xd6aa295b,
0xfe33384a, 0xc000738e, 0xcd67eb2f, 0xe2eb6dc2, 0x97338b02, 0x06c9f246,
0x419cf1ad, 0x2b83c045, 0x3723f18a, 0xcb5b3089, 0x160bead7, 0x5d494656,
0x35f8a74b, 0x1e4e6c9e, 0x000399bd, 0x67466880, 0xb4174831, 0xacf423b2,
0xca815ab3, 0x5a6395e7, 0x302a67c5, 0x8bdb446b, 0x108f8fa4, 0x10223eda,
0x92b8b48b, 0x7f38d0ee, 0xab2701d4, 0x0262d415, 0xaf224a30, 0xb3d88aba,
0xf8b2c3af, 0xdaf7ef70, 0xcc97d3b7, 0xe9614b6c, 0x2baebff4, 0x70f687cf,
0x386c9156, 0xce092ee5, 0x01e87da6, 0x6ce91e6a, 0xbb7bcc84, 0xc7922c20,
0x9d3b71fd, 0x060e41c6, 0xd7590f15, 0x4e03bb47, 0x183c198e, 0x63eeb240,
0x2ddbf49a, 0x6d5cba54, 0x923750af, 0xf9e14236, 0x7838162b, 0x59726c72,
0x81b66760, 0xbb2926c1, 0x48a0ce0d, 0xa6c0496d, 0xad43507b, 0x718d496a,
0x9df057af, 0x44b1bde6, 0x054356dc, 0xde7ced35, 0xd51a138b, 0x62088cc9,
0x35830311, 0xc96efca2, 0x686f86ec, 0x8e77cb68, 0x63e1d6b8, 0xc80f9778,
0x79c491fd, 0x1b4c67f2, 0x72698d7d, 0x5e368c31, 0xf7d95e2e, 0xa1d3493f,
0xdcd9433e, 0x896f1552, 0x4bc4ca7a, 0xa6d1baf4, 0xa5a96dcc, 0x0bef8b46,
0xa169fda7, 0x74df40b7, 0x4e208804, 0x9a756607, 0x038e87c8, 0x20211e44,
0x8b7ad4bf, 0xc6403f35, 0x1848e36d, 0x80bdb038, 0x1e62891c, 0x643d2107,
0xbf04d6f8, 0x21092c8c, 0xf644f389, 0x0778404e, 0x7b78adb8, 0xa2c52d53,
0x42157abe, 0xa2253e2e, 0x7bf3f4ae, 0x80f594f9, 0x953194e7, 0x77eb92ed,
0xb3816930, 0xda8d9336, 0xbf447469, 0xf26d9483, 0xee6faed5, 0x71371235,
0xde425f73, 0xb4e59f43, 0x7dbe2d4e, 0x2d37b185, 0x49dc9a63, 0x98c39d98,
0x1301c9a2, 0x389b1bbf, 0x0c18588d, 0xa421c1ba, 0x7aa3865c, 0x71e08558,
0x3c5cfcaa, 0x7d239ca4, 0x0297d9dd, 0xd7dc2830, 0x4b37802b, 0x7428ab54,
0xaeee0347, 0x4b3fbb85, 0x692f2f08, 0x134e578e, 0x36d9e0bf, 0xae8b5fcf,
0xedb93ecf, 0x2b27248e, 0x170eb1ef, 0x7dc57fd6, 0x1e760f16, 0xb1136601,
0x864e1b9b, 0xd7ea7319, 0x3ab871bd, 0xcfa4d76f, 0xe31bd782, 0x0dbeb469,
0xabb96061, 0x5370f85d, 0xffb07e37, 0xda30d0fb, 0xebc977b6, 0x0b98b40f,
0x3a4d0fe6, 0xdf4fc26b, 0x159cf22a, 0xc298d6e2, 0x2b78ef6a, 0x61a94ac0,
0xab561187, 0x14eea0f0, 0xdf0d4164, 0x19af70ee}
```

# 附录 B

# *E* 函数与加/脱密的伪代码

*E* 函数（输入：in,key1,key2）。

```
1. //利用 3 个临时变量 L,M,R
2. M=in+key1                          //加第 1 个密钥字
3. R=(in<<13)×key2                    //乘第 2 个密钥字，密钥字必须是奇数
4. i=M 的低 9 比特
5. L=S[i]                             //S 盒
6. R=R<<5
7. r=M 的低 5 比特                    //作为循环次数
8. M=M<<r
9. L=L⊕R
10. R=R<<5
11. L=L⊕R
12. r=R 的低 5 比特                   //用于循环次数
13. L=L<<r
14. Output(L,M,R)
MARS-加密(输入：D[ ],K[ ])
```

加密的伪代码如下：

第一部分：前向混乱。

```
1. //子密钥与数据加
2. for i=0 to 3 do
3. D[1]=D[i]=K[i]
4. //前向混乱循环 8 圈
5. for i=0 to 7 do，用 D[0]修改 D[1]、D[2]、D[3]
6. //4 个 S 盒
7. D[1]=D[1]⊕S0[D[0]低字节]
8. D[1]=D[1]+S1[D[0]第二字节]
9. D[2]=D[2]+S0[D[0]第三字节]
10. D[3]=D[3]⊕S1[D[0]高字节]
11. //源字右移位
12. D[0]=D[0]>>24
13. //加乱操作
```

```
14. if i=0 or 4 then
15. D[0]=D[0]+D[3]                    //D[3]加 4 源字
16. if i=0 or 5 then
17. D[0]=D[0]+D[1]                    //D[1]加 4 源字
18. //右循环 D[ ]一个字为下圈做准备
19. (D[3],D[2],D[1],D[0])←(D[0],D[3],D[2],D[1])
20. 结束
```

第二部分：有密钥参与的变换。

```
21. //16 圈有密钥参与变换
22. for i=0 to 15 do
23. (out1,out2,out3)=E(D[0],K[2i+4],K[2i+5])
24. D[0]=D[0]<<13
25. D[2]=D[2]+out2
26. if i<8 then                       //前向模式 8 圈循环
27. D[1]=D[1]+out1
28. D[3]=D[3]⊕out3
29. else                              //后向模式 8 圈循环
30. D[3]=D[3]⊕out1
31. D[1]=D[1]⊕out3
32. end - if
33. //右循环 D[ ]一个字
34. (D[3],D[2],D[1],D[0])←(D[0],D[3],D[2],D[1])
35. end -for 结束
```

第三部分：后向混乱。

```
36. //后向混乱 8 圈
37. for i=0 to 7 do
38. //加乱操作
39. if i=2 or 6 then
40. D[0]=D[0]-D[3]                    //源字减 D[3]
41. if i=3 or 7 then
42. D[0]=D[0]-D[1]                    //源字减 D[1]
43. //4 个 S 盒
44. D[1]=D[1]⊕S1[D[0]低字节]
45. D[2]=D[2]-S0[D[0]高字节]
46. D[3]=D[3]-S0[D[0]第 3 字节]
47. D[3]=D[3]⊕S1[D[0]第 2 字节]
48. //左循环源字
49. D[0]=D[0]<<24
50. D[0]右循环 1 个字
51. (D[3],D[2],D[1],D[0])→(D[0],D[3],D[2],D[1])
```

```
52. end - for
53. //减去密钥
54. for i=0 to 3 do
55. D[i]=D[i]-K[36+i]
```

符号说明：⊕表示逐位异或，∨表示或运算，∧表示与运算，+表示模 $2^{32}$ 加，–表示模 $2^{32}$ 减，×表示模 $2^{32}$ 乘，*a*<<*b* 表示表示将 *a* 左循环移 *b* 位，*a*>>*b* 表示表示将 *a* 右循环移 *b* 位。

脱密伪代码 MARS-decrypt（输入为 *D*[ ],*K*[ ]）如下：

第一部分：前向混乱

```
1. //子密钥与数据加
2. for i=0 to 3 do
3. D[i]=D[i]=K[36+i]
4. //前向混乱循环 8 圈
5. for i=7 down to 0 do
6. //左循环 D[ ]一个字
7. (D[3],D[2],D[1],D[0])←(D[2],D[1],D[0],D[3])
8. //源原字右循环
9. D[0]=D[0]>>24
10. 4 个 S 表
11. D[3]=D[3]⊕S0[D[0]第 2 个字节]
12. D[3]=D[3]+S1[D[0]第 3 字节]
13. D[2]=D[2]+S0[D[0]高字节]
14. D[1]=D[1]⊕S1[D[0]低字节]
15. //附加混乱操作
16. if i=2 或 6，则
17. D[0]=D[0]+D[3]                    //将 D[3]加 4 个源字
18. if i=3 或 7，则
19. D[0]=D[0]+D[1]                    //将 D[1]加 4 个源字
20. for-end
```

第二部分：密钥参与的变换。

```
21. //16 圈有密钥参与变换
22. for i=15 to 0 do
23. //左循环 D[ ]一个字
24. (D[3],D[2],D[1],D[0])←(D[2],D[1],D[0],D[3])
25. D[0]=D[0]>>13
26. (out1,out2,out3)=E 函数(D[0],K[2i+4],K[2i+5])
27. D[2]=D[2]-out2
28. if i<8                          //前向 8 圈
29. D[1]=D[1]-out1
30. D[3]=D[3]⊕out3
31. else                            //后向 8 圈
```

```
32. D[3]=D[3]-out1
33. D[1]=D[1]⊕out3
34. if-end
35. for-end
```

第三部分：后向混乱。

```
36. //后向 8 圈
37. for i=7 to 0 do
38. //D[ ]左循环一个字
39. (D[3],D[2],D[1],D[0])←(D[2],D[1],D[0],D[3])
40. //加乱运算
41. if i=0 或 4，则
42. D[0]=D[0]-D[3]          //D[ ]减源字
43. if i=1 或 5，则
44. D[0]=D[0]-D[1]          //D[1]减源字
45. //源字循环左移
46. D[0]=D[0]<<24
47. 4 个 S 表
48. D[3]=D[3]⊕S1[D[0]高字节]
49. D[2]=D[2]-S0[D[0]第三字节]
50. D[1]=D[1]-S1[D[0]二字节]
51. D[1]=D[1]⊕S0[D[0]低字节]
52. for-end
53. //子密钥减数据
54. for i=0 to 3 do
55. D[i]=D[i]-K[i]
```

# 参 考 文 献


[1] [美]Bruce Schneier．应用密码学：协议、算法与 C 源程序．吴世忠，祝世雄，张文政，等译．北京：机械工业出版社，2013．

[2] 彭川，魏其娇．论分组密码中的数据加密算法 DES．西南民族大学学报（自然科学版），2002(02)．

[3] 于增贵．DES 密码终告被破译．四川省通信学会 1999 年学术年会论文集，1999．

[4] 袁科．俄罗斯密码服务体系研究．贵州大学硕士学位论文，2009．

[5] 戴艺滨．分组密码算法 MISTY1 的分析．信息工程大学，2012．

[6] 张曙港，陈少真．对全轮 Square 算法的低数据 Biclique 攻击．信息工程大学学报，2015(4)．

[7] I.M.Vinogradov.Elements of number theory.Dover Publications Inc., New York, 1945.

[8] Adina Di Porto, William Wolfowicz.VINO : A BLOCK CIPHER INCLUDING VARIABLE PERMUTATIONS.Fondazione Ugo Bordoni, Rome, Italy,1993.

[9] 胡学林．攻破“非线性就性电路密码系统”．通信学报，1993,14(1)．

[10] Guillaume Poupard, Serge Vaudenay. Decorrelated Fast Cipher: An AES Candidate Well Suited for Low Cost Smart Cards Applications.Springer-Verlag.

[11] 吴文玲，李宝，冯登国，等．MAGENTA 的差分密码分析．计算机工程与设计，2000(04)．

[12] 王宏专，杨强浩，李勤，等．MARS 密码简介．网络安全技术与应用，2002(01)．

[13] 何润民，马俊．SHA-256 算法的安全性分析．电子设计工程，2014(03)．

[14] Hans Dobbertin, Antoon Bosselaers, Bart Preneel.RIPEMD-160: A strengthened version of RIPEMD. Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996.

[15] 王可．MD5 算法研究．中文信息，2002(02)．

[16] 梁斌．GSM 系统 A5/1 和 A5/2 加密算法的攻击研究．西安电子科技大学硕士学位论文，2014．

[17] 宋维平．流密码与 RC4 算法．吉林师范大学学报（自然科学版），2005(02)．

[18] 简清明．TwoFish 加密算法及其应用．计算机系统应用，2007(03)．

[19] 宫大力．流密码算法的研究与设计．南京航空航天大学硕士学位论文，2011．

[20] Xuejia Lai, Rainer A. Rueppel.Attacks on the HKM/HFX cryptosystem.Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996．